NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

NAVIGATION SYSTEM DESIGN AND STATE ESTIMATION FOR A SMALL RIGID HULL INFLATABLE BOAT (RHIB)

by

Steven Terjesen

September 2014

Thesis Co-Advisors: Douglas Horner
Sean Kragelund

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 07040–188

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** September 2014	**3. REPORT TYPE AND DATES COVERED** Master's Thesis

4. TITLE AND SUBTITLE
NAVIGATION SYSTEM DESIGN AND STATE ESTIMATION FOR A SMALL RIGID HULL INFLATABLE BOAT (RHIB)

5. FUNDING NUMBERS

6. AUTHOR(S) Steven Terjesen

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES)
N/A

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB protocol number ____N/A____.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE
A

13. ABSTRACT (maximum 200 words)

Autonomous operation of a small rigid hull inflatable boat (RHIB) is a complex problem that requires a robust network of sensors, controllers, processors, and actuators. Furthermore, autonomous navigation requires accurate state estimation, fusing and filtering data from an array of sensors to give the best possible estimates of attitude, position, and velocity.

This thesis will address the hardware modifications and navigation state estimators used to configure the SeaFox Mk II RHIB for future autonomous operations. The study began with a RHIB capable of manual and remote-controlled operation. The proprietary controllers and processors were replaced with an open architecture system that enabled an autonomous mode of operation and data collection from a suite of global positioning satellite receivers and inertial measurement units. Multiple navigation state estimators were designed using the extended Kalman filter and several variants of the unscented Kalman filter. Each filter was evaluated against simulated and actual sea trial data to determine its accuracy, robustness, and computational efficiency.

14. SUBJECT TERMS SEAFOX, RHIB, unmanned, autonomous, extended Kalman filter, EKF, unscented Kalman filter, UKF, square root unscented Kalman Filter, SR-UKF, spherical simplex unscented Kalman filter, SSUKF, square root spherical simplex unscented Kalman filter, SR-SSUKF.	**15. NUMBER OF PAGES** 223
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**
Unclassified	Unclassified	Unclassified	UU

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution unlimited

NAVIGATION SYSTEM DESIGN AND STATE ESTIMATION FOR A SMALL RIGID HULL INFLATABLE BOAT (RHIB)

Steven Terjesen
Lieutenant, United States Navy
B.S., United States Naval Academy, 2008

Submitted in partial fulfillment of the
requirements for the degree of

MECHANICAL ENGINEERING
MASTER OF SCIENCE IN MECHANICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
September 2014

Author: Steven Terjesen

Approved by: Douglas Horner
Thesis Co-Advisor

Sean Kragelund
Thesis Co-Advisor

Garth V. Hobson
Chair, Department of Mechanical & Aerospace Engineering

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Autonomous operation of a small rigid hull inflatable boat (RHIB) is a complex problem that requires a robust network of sensors, controllers, processors, and actuators. Furthermore, autonomous navigation requires accurate state estimation, fusing and filtering data from an array of sensors to give the best possible estimates of attitude, position, and velocity.

This thesis will address the hardware modifications and navigation state estimators used to configure the SeaFox Mk II RHIB for future autonomous operations. The study began with a RHIB capable of manual and remote-controlled operation. The proprietary controllers and processors were replaced with an open architecture system that enabled an autonomous mode of operation and data collection from a suite of global positioning satellite receivers and inertial measurement units. Multiple navigation state estimators were designed using the extended Kalman filter and several variants of the unscented Kalman filter. Each filter was evaluated against simulated and actual sea trial data to determine its accuracy, robustness, and computational efficiency.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

I. INTRODUCTION....1
A. BACKGROUND....1
B. SEAFOX II....2
C. DESIGN REQUIREMENTS....4
D. THESIS OBJECTIVES....5

II. SEAFOX MODIFICATIONS....7
A. OVERVIEW....7
B. SEABORNE CONTROLLER AREA NETWORK....7
C. ADDITIONAL SENSORS....11
D. ROBOT OPERATING SYSTEM....11

III. KALMAN FILTER ESTIMATION REVIEW....13
A. OVERVIEW....13
B. EXTENDED KALMAN FILTER....13
C. UNSCENTED KALMAN FILTER....17
1. Standard UKF....18
2. Square Root UKF....22
3. Spherical Simplex....25

IV. SIMULATION TOOLS AND ASSUMPTIONS....29
A. INTRODUCTION....29
B. SIMULATION ENVIRONMENT AND ASSUMPTIONS....29
1. Simulation Environment....29
2. Simulation Assumptions and Sensor Models....30

V. REFERENCE FRAMES....33
A. OVERVIEW....33
B. REFERENCE FRAME DEFINITIONS....33
1. Earth Centered Inertial....33
2. Earth Centered Earth Fixed....34
3. Geographic (Navigation) Frame....34
4. Body Frame....35
5. Local Tangent Plane....36
C. EARTH REFERENCE ELLIPSOID....36
D. ECEF TO NAVIGATION FRAME TRANSFORMATION....38
E. NAVIGATION TO BODY FRAME....39
F. ECEF TO LOCAL TANGENT PLANE....41

VI. NAVIGATION STATE ESTIMATOR DESIGN....43
A. OVERVIEW....43
B. PROCESS MODEL....43
1. AHRS....45
2. INS Position and Velocity....47
3. SEAFOX II IMU Model....49

C. MEASUREMENT MODELS 53
1. AHRS Measurement Model One 53
2. ARHS Measurement Model Two 59
3. INS Measurement Model 62
D. EKF EQUATIONS 63
1. AHRS 63
2. INS 66
E. UKF 67

VII. DISCUSSION AND RESULTS 69
A. OVERVIEW 69
B. FILTER TUNING 69
C. FILTER COMPUTATION SPEED COMPARISON 72
D. ATTITUDE PERFORMANCE 75
E. POSITION AND VELOCITY PERFORMANCE 87
F. ESTIMATOR RANKING AND SELECTION 89
G. SEAFOX II ESTIMATION COMPARISON 95

VIII. CONCLUSION AND RECOMMENDATIONS 101
A. CONCLUSION 101
B. RECOMMENDATIONS FOR FUTURE WORK 102

APPENDIX: MATLAB CODE 105
A. OVERVIEW 105
B. EXTENDED KALMAN FILTER MATLAB CODE 105
1. Measurement Model One Implementation 105
2. Measurement Model Two Implementation 114
C. STANDARD UNSCENTED KALMAN FILTER MATLAB CODE 125
1. Measurement Model One Implementation 125
2. Measurement Model Two Implementation 134
D. SQUARE ROOT UNSCENTED KALMAN FILTER MATLAB CODE 142
1. Measurement Model One Implementation 142
2. Measurement Model Two Implementation 152
E. SPHERICAL SIMPLEX UNSCENTED KALMAN FILTER MATLAB CODE 161
1. Measurement Model One Implementation 161
2. Measurement Model Two Implementation 171
F. SQUARE ROOT SPHERICAL SIMPLEX UNSCENTED KALMAN FILTER MATLAB CODE 181
1. Measurement Model One Implementation 181
2. Measurement Model Two Implementation 192

LIST OF REFERENCES 203

INITIAL DISTRIBUTION LIST 205

LIST OF FIGURES

Figure 1. SEAFOX I in low speed (left) and high speed (right) operation, from [2].....2
Figure 2. SEAFOX II prior to modification. 2
Figure 3. Legacy SEAFOX II control input architecture. 3
Figure 4. Mounting and operation of the ATLAS Sonar 4
Figure 5. High speed maneuverable surface target (top), ship deployable surface target—Jet Ski (bottom), from [3]. 8
Figure 6. King node system status display. 9
Figure 7. Several of the nodes and sensors installed on the SEAFOX II. 10
Figure 8. EKF recursion summary. 16
Figure 9. The unscented transformation of an arbitrary nonlinear function, after [13]. 17
Figure 10. Standard UKF recursion. 22
Figure 11. SR-UKF recursion. 25
Figure 12. 3rd order filter for GPS acceleration estimation. 31
Figure 13. Navigation, ECEF, and ECI coordinate reference frames, after [19]. 35
Figure 14. Body reference frame representation, from [8].....36
Figure 15. Earth reference ellipsoid, after [8]. 37
Figure 16. Plane rotations of yaw, pitch, and roll from navigation to body.....40
Figure 17. Cascaded Kalman filter for state estimation. 44
Figure 18. Measurements from the IMU accelerometer channels. 50
Figure 19. Measurements from the IMU gyro roll rate, pitch rate, and yaw rate channels.....52
Figure 20. Roll measurements as approximate from MM1.....58
Figure 21. Pitch measurements as approximate from measurement model 1. 58
Figure 22. Acceleration X-channel comparison using the equations from MM2. 61
Figure 23. Acceleration Y-channel comparison using the equations from MM2. 61
Figure 24. Acceleration Z-channel comparison using the equations from MM2. 62
Figure 25. Comparison of relative computation time of a 10 seconds data set run in Simulink against the EKF(1) run time. 73
Figure 26. Comparison of relative computation time for a single iteration of each estimator function against EKF(1). 74
Figure 27. Roll angle estimates during a high rate turn with MM1 (top), MM2 (bottom).....77
Figure 28. Roll angle estimates with MM1 (top), MM2 (bottom). 78
Figure 29. Pitch angle estimates during a high rate turn with MM1 (top), MM2 (bottom).....79
Figure 30. Pitch angle estimates with MM1 (top), MM2 (bottom).....80
Figure 31. Roll angle estimates for MM1 estimators when gyro and accelerometer biases are unknown during a high rate turn (top), MM2 (bottom). 81
Figure 32. Roll angle estimates for MM1 estimators when gyro and accelerometer biases are unknown (top), MM2 (bottom). 82

Figure 33. Pitch angle estimates for MM1 estimators when gyro and accelerometer biases are unknown during a high rate turn (top), MM2 (bottom). 83
Figure 34. Pitch angle estimates for measurement model 2 estimators when gyro and accelerometer biases are unknown (top), MM2 (bottom).....84
Figure 35. SR-UKF gyro bias estimates for experiment 2. 85
Figure 36. Summary of averaged RMS error for attitude estimation for both experiments. 86
Figure 37. RMS velocity estimate errors.....87
Figure 38. Accelerometer RMS bias errors for SR-UKF(2) with arbitrary initial bias values. 88
Figure 39. RMS position estimate errors.....89
Figure 40. Radar plot of filter evaluation criteria, score on a scale of one to nine, where nine is the best score. 90
Figure 41. Weighted estimator comparison. 93
Figure 42. Weighted estimator score based on RMS errors and design weights. Lowest score is best performing filter. 94
Figure 43. EKF(1) and SR-SSUKF(1) position estimate comparison during turning circle maneuvers. 95
Figure 44. EKF(1) and SR-SSUKF(1) velocity estimate comparison during turning circle maneuvers. 96
Figure 45. EKF(1) and SR-SSUKF(1) roll estimates compared against unfiltered gyro integration. 97
Figure 46. EKF(1) and SR-SSUFK(1) pitch estimates compared against an unfiltered gyro solution. 98

LIST OF TABLES

Table 1. List of nodes installed on SEAFOX II. 9
Table 2. Honeywell HG1700AG58 sensor specifications, from [5]. 11
Table 3. Values for simulated sensor PSD and bias. 32
Table 4. WGS84 parameter constants, from [8]. 37
Table 5. Noise characteristics for the SEAFOX II GPS sensor, after [4]. 63
Table 6. Noise variance values for process and measurement noise in the Condor simulation environment. 70
Table 7. Final process and measurement noise scaling factors 71
Table 8. Number of iterations per time step of each UKF variant for the given estimator design. 72
Table 9. Initial state and covariance estimates used initialize each navigation estimator. 76
Table 10. Subjective weights assigned to each evaluated estimation category. 92

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

AHRS	attitude, heading, and reference system
ASW	anti-submarine warfare
AUV	autonomous underwater vehicle
CAN	controller area network
CAVR	Center for Autonomous Vehicle Research
DCM	direction cosine matrix
EKF	extended Kalman filter
EW	electronic warfare
GPS	global positioning satellite
HSMST	high speed maneuverable surface target
IMU	inertial measurement unit
INS	inertial navigation system
ISR	intelligence, surveillance, and reconnaissance
MCM	mine countermeasures
MIO	maritime interdiction operations
MM1	measurement model one
MM2	measurement model two
MS	maritime security
PID	proportional-integral-derivative
PSD	power spectral density
RHIB	rigid hull inflatable boat
RMS	root mean squared
SEACAN	seaborne controller area network
SOF	special operations forces
SPU	signal processing unit
SR-SSUKF	square root spherical simplex unscented Kalman Filter
SR-UKF	square root unscented Kalman filter
SSUKF	spherical simplex unscented Kalman filter
SUW	surface warfare
UKF	unscented Kalman filter

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

Foremost, I would like to thank my advisors, Sean Kragelund and Professor Doug Horner, as well as Aurelio Monarrez for all the work they put in to the SEAFOX II and extra hours spent on the water collecting data for my work.

I would like to thank David Purdy, Bill McAuley, and their support team from the Port Hueneme Surface Targets Group for all the help and guidance in transforming the SEAFOX II. Your contribution to this work and future research at NPS is immeasureable.

Professors Issac Kaminer and Vladimir Dobrokhodov, thank you for helping lay the foundation by which this thesis was written, I would have never gotten this far without your help.

Lastly, to my family, thank you for supporting me in all that I do.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. BACKGROUND

The United States Navy's *Unmanned Surface Vehicle Master Plan* defines the Navy's long term commitment to the research and development of unmanned surface vehicles. There are seven critical mission areas the Navy intends to support with unmanned surface craft: mine countermeasures (MCM), anti-submarine warfare (ASW), maritime security (MS), surface warfare (SUW), Special Operations Forces (SOF) support, electronic warfare (EW), and maritime interdiction operations (MIO) support [1]. Each warfare mission area requires unique system configurations and demands, but share common ties to the problem of open water navigation and obstruction avoidance.

The Naval Postgraduate School Center for Autonomous Vehicle Research (CAVR) has been developing surface and subsurface vehicles to navigate unknown waterways with semi and full autonomy with aims to support SOF and intelligence, surveillance, and reconnaissance (ISR) missions. These vessels use various sensors to collect data from the surrounding environment to make navigation decisions and avoid surface and subsurface obstacles.

The SEAFOX I (Figure 1), built by Northwind Marine, Inc. and modified by previous thesis students and faculty, was CAVR's first attempt at full surface vessel autonomy. SEAFOX I has been used to conduct experiments for autonomous riverine navigation, networked collaborative multi-vehicle operations, and maritime interdiction operations. While the vessel has seen much success, it suffers from significant control command to execution lag due to legacy and proprietary controller interfaces. This thesis and future work will focus on building upon the successes of the SEAFOX I while leveraging new technologies to avoid previous control limitations on a new platform for autonomous surface vessel (ASV) research.

Figure 1. SEAFOX I in low speed (left) and high speed (right) operation, from [2].

B. SEAFOX II

The SEAFOX II (see Figure 2), designed and built by Northwind Marine, Inc., is a 5.1 meter rigid hull inflatable boat (RHIB). Its water jet propulsion system is powered by a JP-5 fueled Mercury 3.0 liter V-6 OptiMax JP racing engine capable of sustained speeds over 30 knots. The water jet acts as both propulsion and rudder giving the vessel good performance at high speed but poor maneuverability at low speed.

Figure 2. SEAFOX II prior to modification.

Northwind Marine designed the SEAFOX II for two modes of operation: manual and remote. In manual operation mode the boat is controlled by an onboard operator through the helm and throttle lever or a digital autopilot control display. For remote operation the SEAFOX II is outfitted with a wireless communications suite that relays commands and data between the vessel and a remote control station. In both manual autopilot and remote control operation, all control signals are routed through a proprietary signal processor unit (SPU) for relay to the rudder and throttle actuators. Figure 3 depicts the legacy SEAFOX II architecture from control input to output. Prior to modification, the SEAFOX II provided no organic means of capturing rudder feedback, throttle position, engine RPM, GPS location or speed for online or offline data analysis. The SPU and associated controllers were locked in a proprietary system leaving few options for experimental methods of control.

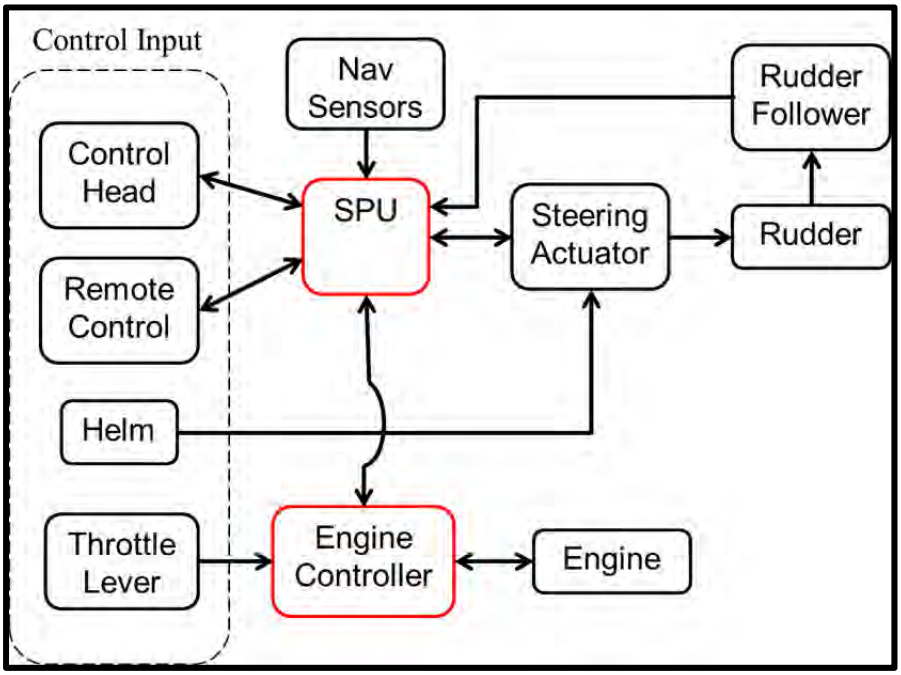

Figure 3. Legacy SEAFOX II control input architecture.

As the SEAFOX II held little capability for autonomous control experimentation, it was used primarily as a designated chase boat for the SEAFOX I ASV as well as a

launch/recovery platform for Hydroid REMUS 100 autonomous underwater vehicles (AUVs).

C. DESIGN REQUIREMENTS

In support of ongoing research programs, the SEAFOX II is being modified to deploy the autonomous topographic large area survey (ATLAS) sonar system designed by the Applied Research Laboratories of the University of Texas at Austin (ARL:UT). The ATLAS sonar will be mounted to the bow via an electrically-actuated truss designed by Hullux Subsea Technologies to deploy the sonar below the waterline, as shown in Figure 4.

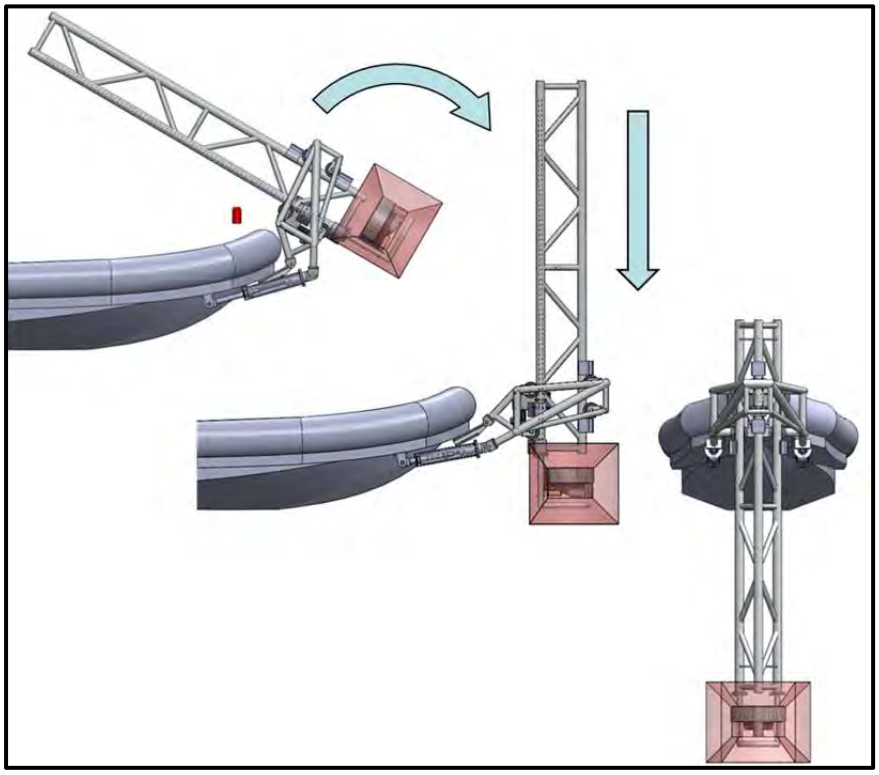

Figure 4. Mounting and operation of the ATLAS Sonar

The ATLAS is designed to detect subsurface obstacles, particularly in the riverine and littoral environment. Aside from obstacle detection, the sonar will provide local bathymetry data such as water depth, temperature, and sound speed velocity. In order for

the sonar to provide accurate information it requires the following data inputs at an update rate of 10 Hz:

- Platform navigation time: The reference time of the vehicle for all parameters in the current navigation data sample.
- Geographic latitude and longitude.
- Pitch angle: Positive angles are measured upward from the horizontal plane.
- Roll angle: Positive angles are measured clockwise when viewed from the platform bow.
- Heading angle: Positive angles are measured clockwise from true north.
- North and east velocity components.
- Platform water speed: Speed of the platform relative to the water in the direction of the current heading.

Beyond providing quality navigation data to the ATLAS sonar, further modifications needed to be made to the existing control network to advance the progress towards full autonomy. To allow for a more complete research vessel and to circumvent the existing proprietary controller network, the SEAFOX II needed to be outfitted with a new control architecture and sensor suite capable of the following:

- An open architecture network capable of scheduling tasks, executing orders, and logging data from multiple controllers, actuators, and sensors.
- Three modes of operational control: manual, remote controlled, and fully autonomous.
- Inertial and GPS data.
- Engine control and feedback.
- Rudder control and feedback.
- Wireless communications for remote control operation and monitoring.

D. THESIS OBJECTIVES

To meet the design requirements, this thesis focused on two major objectives: first, to modify the SEAFOX II hardware to provide the ability to capture and store live GPS and inertial measurement data, and secondly to investigate methods of determining the vessel's attitude, velocity, and position at a high update rate in support of both sonar and future autonomous operations. In pursuit of the second objective a navigation state

estimator needed to be designed. For this task, five Kalman filter variants for state estimation were investigated for accuracy, robustness, and computational efficiency: the extended Kalman filter (EKF), unscented Kalman filter (UKF), spherical simplex unscented Kalman filter (SSUKF), square root unscented Kalman filter (SR-UKF), and the square root spherical simplex unscented Kalman filter (SR-SSUKF). Each estimator was evaluated using simulated data and the best performing estimator was evaluated against data obtained from the SEAFOX II while operating on the Monterey Bay.

II. SEAFOX MODIFICATIONS

A. OVERVIEW

Building a sensor and controller network for an autonomous vehicle from scratch has potential to be a lengthy and costly process depending on the state of the baseline vehicle. The SEAFOX II provided a unique platform that was originally designed from the ground up with unmanned operation in mind. All of the actuators necessary to operate the rudder and throttle were already integrated into the baseline system. While this thesis focused on modifications specific to the SEAFOX II, the implemented system architecture has been demonstrated to work on a variety of vessels. The simplicity of this architecture enabled its complete installation onboard SEAFOX II in under one week.

B. SEABORNE CONTROLLER AREA NETWORK

The seaborne controller area network (SEACAN) was developed by a joint government and contractor team managed by the Surface Targets Branch, Naval Air Warfare Center Weapons Division (NAWCWD) at Naval Base Ventura County, Port Hueneme, California. It is a scalable, open architecture, root level access system that permitted CAVR to quickly upgrade the the SEAFOX II autonomous capabilities.

The Surface Targets Group develops, builds, and maintains a growing fleet of unmanned surface vessel targets, two examples of which are shown in Figure 5. The vessel by which the SEAFOX II modifications based is the high speed maneuverable surface target (HSMST), a 7 meter, twin outboard USV with a top speed of 45 knots [3]. The HSMST is remote operated from a base station and is capable of autopilot waypoint tracking. The HSMST class of vessel was designed and built with ease of replication, low cost, and robustness in mind as these targets are often destroyed by the naval weapon systems that engage them.

Figure 5. High speed maneuverable surface target (top), ship deployable surface target—Jet Ski (bottom), from [3].

The controller area network (CAN) is the automotive serial bus standard for intra-vehicular device communication, connecting a network of nodes that operate sensors, actuators, and other control devices without the need of a central host computer. The SEACAN system is based on the architecture of the CAN, adapted for use in the maritime environment. The nodes the make up the SEACAN system are listed in Table 1. Each node, with the exception of the king and port and starboard engine actuator nodes,

contains a central processing unit, 4th generation (CPU-4) microcontroller board for sensor and actuator control.

Node List
• King • Rudder Feedback • Rudder Controller • Compass & Heading • RF Modem • GPS • Engine Controller (Port & Starboard) • Throttle Controller (Port & Starboard)

Table 1. List of nodes installed on SEAFOX II.

The king node provides the monitoring and supervision of the vessel's controller area network (CAN) bus activity and indicates system status via an LCD display as shown in Figure 6. The display not only shows node status such as heading or rudder positions but also has various options for tuning the system such as resetting the rudder's centerline zero position.

Figure 6. King node system status display.

The remainder of the nodes listed in Table 1 function as their name implies. Several nodes are depicted in Figure 7. The rudder controller regulates the single speed hydraulic steering pump with pulse width modulation to allow for variable motor speeds. The rudder controller utilizes a tuned proportional-integral-derivative controller with set saturation limits to insure smooth rudder movements. The rudder feedback provides current rudder angle information. The heading node relays Euler angles and rates provided by the MicroStrain 3DM-GX1 inertial measurement unit (IMU). When operating in remote mode, the RF modem node sends periodic information to the base station operator and receives commands as necessary. For safety purposes, if the carrier signal is lost for a specified amount of time, the vessel will enter a holding pattern while awaiting reconnection. If reconnection with the base station is not regained within a specified time window the kill switch is enabled and the engines shutdown. The engine controller nodes monitor the engine RPM and control the engine startup and shutdown while the throttle controller monitors the throttle position and executes throttle position commands.

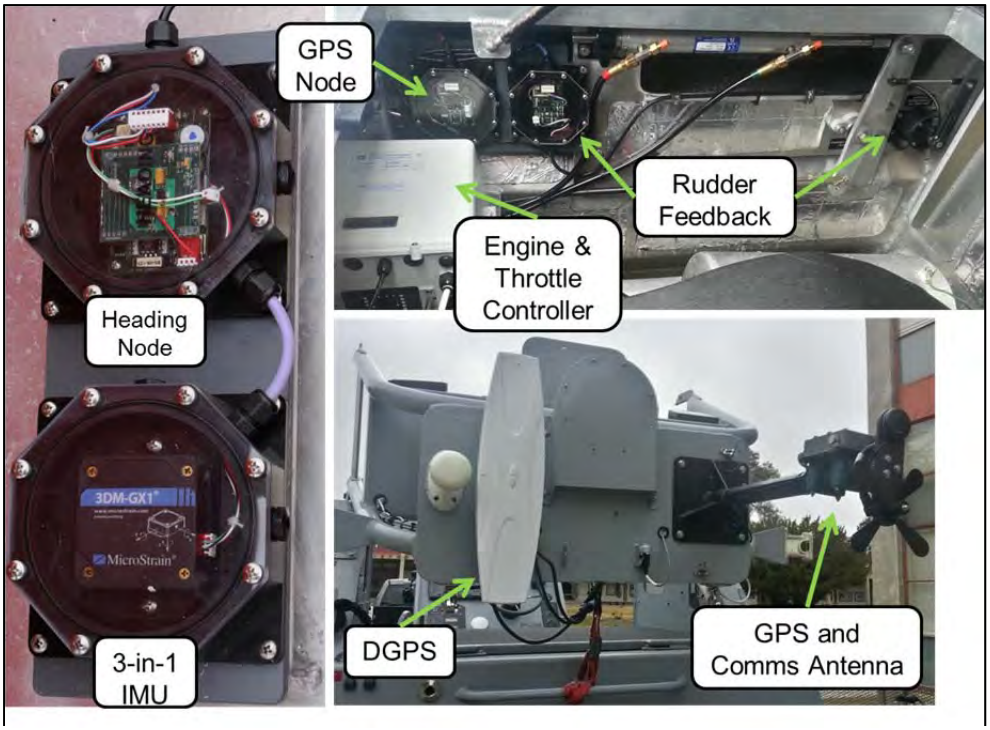

Figure 7. Several of the nodes and sensors installed on the SEAFOX II.

C. ADDITIONAL SENSORS

In addition to the sensors organic to the SEACAN system, an additional high performance IMU and GPS receiver were installed. The IMU installed is a Honeywell HG1700AG58 containing a three-axis ring laser gyro (RLG) and accelerometer. The IMU sensor specifications are listed in Table 2. The IMU takes readings at 600 Hz and outputs to a standard RS-422 serial interface at a rate of 100 Hz. The additional GPS receiver, a ComNav Vector G2 GPS Satellite Compass, shown as DGPS in Figure 7, houses two GPS receivers 0.5 meters apart as well as a two-axis gyro. The DGPS outputs position, heading, turn rate, pitch or roll, course over ground, and speed over ground at a rate of 20 Hz. The ComNav GPS has 5 meter position accuracy and 0.5 degree heading accuracy [4].

Honeywell HG1700AG50 Sensor Specifications							
Sensor	**Operating Range**	**Scale Factor**	**Scale Factor Accuracy**	**Bias**	**Axis Alignment**	**Output Noise**	**Frequency Response**
Accel.	37 (+/- g's)	2^{1-4} x 600 (ft/sec$^{2)}$	1000 (ppm, 1 sigma)	50 (milli-g's, 1 sigma)	0.5 (milli-rad, 1 sigma)	100 (milli-g's max)	70 max. (90 deg phase lag at freq in Hz)
Gyro.	1074 (+/- deg/sec)	2^{2-0} x 600 (rad/sec)	1000 (ppm, 1 sigma)	50 (deg/hr, 1 sigma)	0.5 (milli-rad, 1 sigma)	45 (milli-rads/sec max)	70 max. (90 deg phase lag at freq in Hz)

Table 2. Honeywell HG1700AG58 sensor specifications, from [5].

D. ROBOT OPERATING SYSTEM

An added benefit to the SEACAN structure is the ability to issue rudder and engine commands from outside the system. This plug and play capability allows for the development of an autonomous mode of operation, controlled by an onboard computer. The Robot Operating System (ROS) was used to develop the software to enable data

acquisition and of the SEAFOX II. ROS is a widely supported, open source, open architecture operating system that provides a large library of tools associated with robot control and communication [6]. ROS runs on SEAFOX II from an embedded PC-104 and executes programs that store sensor data from the SEACAN system and other sensors in a common database, run the designated navigation filters, and issue commands to the SEACAN rudder and throttle controllers. Once the ATLAS sonar suite is installed, programs that analyze sonar data, generate navigation paths, and follow those paths will be created to enable a fully autonomous mode of operation.

III. KALMAN FILTER ESTIMATION REVIEW

A. OVERVIEW

The Kalman filter, developed by [7], is a powerful estimator used in a wide range of applications from computer vision to navigation. For linear systems, the Kalman filter has been proven to provide an optimal solution to minimize the error covariance between the true state and its estimate [8]. For non-linear systems, the extended Kalman filter (EKF) is a commonly used method for state estimation, but provides sub-optimal error covariance minimization due to its first order linearization of non-linear processes. Many alternatives to the classic EKF have been developed that vary in computational complexity and cost. In this paper, the unscented Kalman filter (UKF) and several variants are explored as an alternative to the EKF in order to make use of the highly nonlinear measurement equations used in the subsequent state estimator design.

B. EXTENDED KALMAN FILTER

The extended Kalman filter has been studied in depth with numerous publications available deriving and analyzing the process. This section uses the equations derived in [9] and chapters 4 and 5 of [8] to summarize the EKF process.

The discrete time extended Kalman filter addresses the issue of estimating the true state $x \in R^n$ driven by the nonlinear process

$$x_{k+1} = f\left(x_k, u_k, v_k\right) \tag{3.1}$$

where u_k is a process input measurement and v_k is process noise at time instance k.

The process nonlinear measurements $z \in R^m$ are defined by

$$z_k = h\left(x_k, w_k\right) \tag{3.2}$$

where w_k is the measurement noise at time instance k.

The process and measurement noise is assumed to be zero mean Gaussian, independent, and uncorrelated white noise:

$$\begin{aligned} v_k &\sim N(0, Q) \\ w_k &\sim N(0, R) \end{aligned} \tag{3.3}$$

The estimate of the state at time k is shown as $\hat{x}_k$. The nonlinear state time propagation equation $\tilde{x}_{k+1}$ and measurement process $\tilde{z}_k$ are defined as

$$\begin{aligned} \tilde{x}_{k+1} &= f(\hat{x}_k, u_k, 0) \\ \tilde{z}_k &= h(\tilde{x}_k, 0) \end{aligned} \tag{3.4}$$

which are approximations of the state and measurement with noise terms set to their expected value of zero.

By linearizing the non-linear functions in Equation (3.4) around the current value of the state estimate, the Kalman filter solution becomes

$$\begin{aligned} x_{k+1} &\approx \tilde{x}_k + F_k(x_k - \hat{x}_k) + V_k v_k \\ z_k &\approx \tilde{z}_k + H_k(x_k - \tilde{x}_k) + W_k w_k \end{aligned} \tag{3.5}$$

where

$$\begin{aligned} F_{ij} &= \left.\frac{\partial f_i}{\partial x_j}\right|_{(\hat{x}_k, u_k, 0)}, V_{ij} = \left.\frac{\partial f_i}{\partial v_j}\right|_{(\hat{x}_k, u_k, 0)} \\ H_{ij} &= \left.\frac{\partial h_i}{\partial x_j}\right|_{(\hat{x}_k, 0)}, W_{ij} = \left.\frac{\partial h_i}{\partial w_j}\right|_{(\hat{x}_k, 0)} \end{aligned} \tag{3.6}$$

are the first order Jacobians of the nonlinear state process, state excitation process, measurement process, and measurement noise process, respectively.

By defining an error process as the difference between the true and estimated values

$$\begin{aligned} \tilde{e}_{x_{k+1}} &= x_{k+1} - \tilde{x}_{k+1} \\ \tilde{e}_{z_k} &= z_k - \tilde{z}_k \end{aligned} \tag{3.7}$$

with an associated covariance of

$$P_k \sim N\left(0, E\left[\tilde{e}_{x_k} \tilde{e}_{x_k}^T\right]\right) \tag{3.8}$$

where $E[\bullet]$ is the expected value operator.

The process and measurement noise processes are now defined as

$$\begin{aligned} \tilde{Q}_k &\sim N\left(0, V_k Q V_k^T\right) \\ \tilde{R}_k &\sim N\left(0, W_k R W_k^T\right) \end{aligned} \tag{3.9}$$

Equation (3.5) can then be substituted into Equation (3.7) to form

$$\begin{aligned} \tilde{e}_{k+1} &= F\left(x_k - \hat{x}_k\right) + V_k v_k \\ \tilde{e}_{z_k} &= H\left(\tilde{e}_{x_k}\right) + W_k w_k \end{aligned} \tag{3.10}$$

making the error process linear with respect to the state error.

Since the actual measurement at time k is known, it is possible to use the measurement error $\tilde{e}_{z_k}$ to estimate the predicted state error $\tilde{e}_{x_k}$ and correct an *a posteriori* state estimate

$$\hat{x}_k = \tilde{x}_k + K_k \tilde{e}_{z_k} \tag{3.11}$$

The Kalman gain K_k minimizes the error covariance P_k. The Kalman gain is calculated at each time instance by the equation

$$K_k = P_k^- H_k \left(H_k P_k^- H_k^T + \tilde{R}_k\right)^{-1} \tag{3.12}$$

where the "-" superscript denotes an *a priori* estimate.

The *a posteriori* state estimate is rewritten to take the form

$$\hat{x}_k^+ = \hat{x}_k^- + K_k\left(z_k - h\left(\hat{x}_k^-, 0\right)\right) \tag{3.13}$$

where the "+" superscript denotes the corrected *a posteriori* state estimate.

The error state covariance is then updated as

$$P_k^+ = \left(I - K_k H_k\right) P_k^- \tag{3.14}$$

The state and error covariance estimates are projected ahead to the next time instance by

$$\hat{x}_{k+1}^- = f\left(\hat{x}_k^+, u_k, 0\right) \tag{3.15}$$

$$P_{k+1}^{-} = F_k P_k^{+} F_k^{T} + \tilde{Q}_k \tag{3.16}$$

The EKF recursion is executed at every time instance as shown in Figure 8. The Kalman gain is computed with the time updates from the previous time step. The *a priori* state estimate and error state is then corrected providing the *a posteriori* estimates which are propagated through their nonlinear transformations to provide a time update.

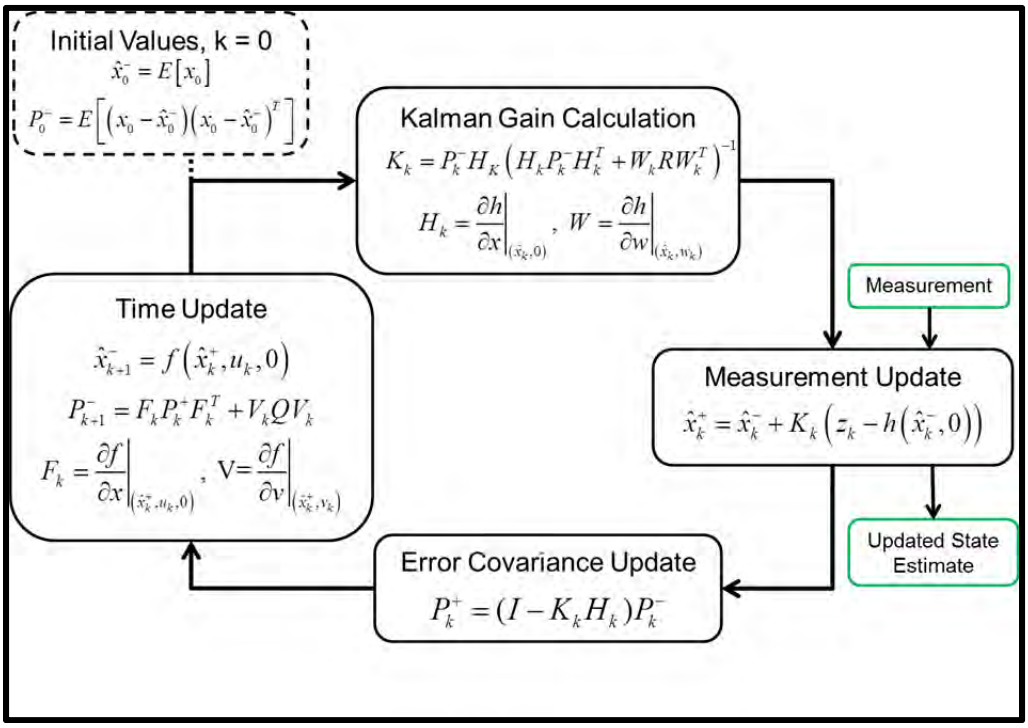

Figure 8. EKF recursion summary.

The performance of the EKF is tightly coupled to the nonlinearity of the system process and measurements. The EKF state distribution is approximated by a Gaussian random variable which is propagated through a first-order linearization of the nonlinear process. If the error propagation cannot be accurately approximated with a linearized function due to the function's higher order terms, then the estimates will be poor and may diverge. An additional source of error is the difficulty of developing and coding the Jacobian matrices from Equation (3.6) which can be highly complex and extensive depending on the nonlinear process and measurement functions. Higher order filtering schemes such as Bayesian or Particle filters address the issues associated with linearization by approximating the true probability density function with large numbers

of random sample points, but at a significant computational cost. For further analysis of the Bayesian filter see [10]. As the state estimators developed for this thesis are intended for testing in a real-time system, a less computationally burdensome filter is required.

C. UNSCENTED KALMAN FILTER

The unscented Kalman Filter (UKF), first published in [11], is founded on the idea that it is simpler to approximate a probability distribution function than an arbitrary nonlinear function. By propagating a finite set of specifically chosen points (sigma points), with a given mean and covariance, through a nonlinear function to form a cloud of transformed points, the statistics of the transformed points can be estimated to approximate the true mean and covariance as depicted in Figure 9. For Gaussian inputs, the UKF approach is accurate to the third order Taylor series expansion of a nonlinear function, and for non-Gaussian inputs the approximations are accurate to the second order, whereas the linearization approach of the EKF is only accurate to the first order [12].

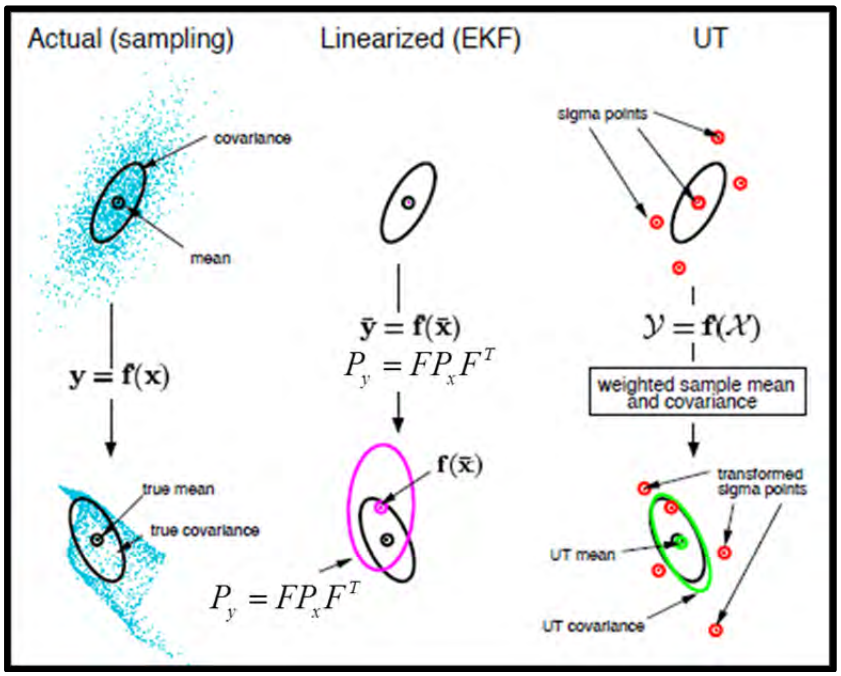

Figure 9. The unscented transformation of an arbitrary nonlinear function, after [13].

Several variants of unscented transformations have been developed for the purpose of computational savings such as the square root UKF (SR-UKF) [14], spherical simplex UKF (SSUKF) [15], and the square root spherical simplex UKF (SR-SSUKF) [16]. The algorithms for each variant, including the standard UKF implementation will be developed in subsequent sections.

1. Standard UKF

Following the unscented transformation (UT) algorithms from [13], consider a random variable $x \in R^{L\times 1}$, where L is the number of states being estimated, that is propagated through an arbitrary nonlinear function $y = h(x)$. If x has an assumed mean of $\bar{x}$ and covariance P_x, the statistics of y can be calculated by first forming a matrix χ of $2L+1$ sigma vectors according to

$$\begin{aligned} \chi_0 &= \bar{x} \\ \chi_i &= \bar{x} + \left(\sqrt{(L+\lambda)P_x}\right)_i \qquad i = 1,\ldots,L \\ \chi_i &= \bar{x} - \left(\sqrt{(L+\lambda)P_x}\right)_{i-L} \qquad i = L+1,\ldots,2L \end{aligned} \tag{3.17}$$

where λ is a scaling parameter calculated by

$$\lambda = \alpha^2 (L+\kappa) - L \tag{3.18}$$

The parameter α determines the spread of the sigma points around the mean $\bar{x}$ and is a tunable parameter that is generally set to values $1\times 10^{-4} \le \alpha \le 1$. The parameter κ is a secondary scaling parameter that is generally set to values of $\kappa = 3 - L$ or zero. The term $\left(\sqrt{(L+\lambda)P_x}\right)_i$ in Equation (3.17) represents the ith row or column of the matrix square root depending on the function used to calculate it. Using MATLAB's *chol* function to calculate the matrix square root leads to using the ith row.

Next, the sigma vectors from Equation (3.17) are individually passed through the nonlinear function to form

$$\Upsilon_i = h\left(\chi_i\right) \quad i = 0,...,2L \tag{3.19}$$

Lastly, the mean and covariance for y are approximated using a weighted sample mean and covariance of the transformed sigma points by

$$\bar{y} = \sum_{i=0}^{2L} W_i^{(m)} \Upsilon_i \tag{3.20}$$

$$P_y = \sum_{i=0}^{2L} W_i^{(c)} \left(\left[\Upsilon_i - \bar{y}\right]\left[\Upsilon_i - \bar{y}\right]^T \right) \tag{3.21}$$

The sigma weights are calculated according to:

$$\begin{aligned} W_0^{(m)} &= \frac{\lambda}{L+\lambda} \\ W_0^{(c)} &= \frac{\lambda}{L+\lambda} + (1-\alpha^2+\beta) \\ W_i^{(m)} &= W_i^{(c)} = \frac{1}{2\left(L+\lambda\right)} \quad i = 0,...,2L \end{aligned} \tag{3.22}$$

where β is determined by the assumed probability distribution of x. For Gaussian distributions $\beta = 2$ is the optimal value. The UT can be extended to the problem of state estimation, forming the unscented Kalman filter.

For a state vector $x \in R^{n\times 1}$ where *n* is the number of states being estimated, let the nonlinear system given by

$$\begin{aligned} x_{k+1} &= f\left(x_k, u_k, v_k\right) \\ z_k &= h\left(x_k, w_k\right) \\ v_k &\sim N\left(0, Q\right) \\ w_k &\sim N\left(0, R\right) \end{aligned} \tag{3.23}$$

Let an augmented state estimation vector of dimension L be defined as

$$\hat{x}_{k-1}^a = \begin{bmatrix} \hat{x}_{k-1}^T & v_{k-1}^T & w_{k-1}^T \end{bmatrix}^T \tag{3.24}$$

where the augmented state vector is extended to include all measurement and process noise terms.

The state error covariance matrix is also extended to include the measurement and process noise matrices:

$$P_{k-1}^{a} = \begin{bmatrix} P_{k-1}^{+} & 0 & 0 \\ 0 & Q & 0 \\ 0 & 0 & R \end{bmatrix} \tag{3.25}$$

The $2L+1$ sigma vectors are calculated following Equation (3.17) :

$$\begin{gathered} \chi_{k-1}^{a} = \begin{bmatrix} \hat{x}_{k-1} & \hat{x}_{k-1} + \sqrt{(L+\lambda)P_{k-1}^{a}} & \hat{x}_{k-1} - \sqrt{(L+\lambda)P_{k-1}^{a}} \end{bmatrix} \\ \chi^{a} = \begin{bmatrix} \left(\chi^{x}\right)^{T} & \left(\chi^{Q}\right)^{T} & \left(\chi^{R}\right)^{T} \end{bmatrix}^{T} \end{gathered} \tag{3.26}$$

The sigma vectors are then propagated through the nonlinear system process defined in Equation (3.23) using the weights from Equation (3.22) to calculate the *a priori* transformed mean and covariance of the state and measurement estimates according to:

$$\chi_{k}^{x} = f\left(\chi_{k-1}^{x}, \chi_{k-1}^{Q}, u_{k}\right) \tag{3.27}$$

The 2*L*+1 estimates are used to create a weighted mean value for the state at time *k*:

$$\hat{x}_{k}^{-} = \sum_{i=0}^{2L} W_{i}^{(m)} \chi_{i,k}^{x} \tag{3.28}$$

The covariance of the transformed process is calculated by executing a weighted expected value operation between the transformed sigma points and the weighted mean:

$$P_{k}^{-} = \sum_{i=0}^{2L} W_{i}^{(c)} \left[\chi_{i,k}^{x} - \hat{x}_{k}^{-}\right]\left[\chi_{i,k}^{x} - \hat{x}_{k}^{-}\right]^{T} \tag{3.29}$$

Similarly, the transformed sigma points are used in the measurement process to calculate the weighted mean value of the measurement estimate according to:

$$\begin{aligned} \Upsilon_{k} &= h\left(\chi_{k}^{x}, \chi_{k}^{R}\right) \\ \hat{z}_{k} &= \sum_{i=0}^{2L} W_{i}^{(m)} \Upsilon_{i,k} \end{aligned} \tag{3.30}$$

The covariance of the measurement process is estimated by executing a weighted expected value between the transformed measurement estimates and weighted mean of those estimates:

$$P_{\hat{z}_k,\hat{z}_k} = \sum_{i=0}^{2L} W_i^{(c)} \left[\Upsilon_{i,k} - \hat{z}_k \right] \left[\Upsilon_{i,k} - \hat{z}_k \right]^T \tag{3.31}$$

The cross covariance between the state process and measurement process is approximated as the weighted covariance:

$$P_{\hat{x}_k,\hat{z}_k} = \sum_{i=0}^{2L} W_i^{(c)} \left[\chi_{i,k}^x - \hat{x}_k^- \right] \left[\Upsilon_{i,k} - \hat{z}_k \right]^T \tag{3.32}$$

With the time update complete and covariance determined, the state corrections can now be implemented via the standard Kalman filter measurement update equations. The Kalman gain, however, is calculated using the cross covariance and measurement covariance instead of solving the algebraic Riccati equation as in the EKF implementation:

$$K_k = P_{\hat{x}_k,\hat{z}_k} P_{\hat{z}_k,\hat{z}_k}^{-1} \tag{3.33}$$

The state and state covariance is lastly corrected in the same manner as the EKF:

$$\begin{aligned} \hat{x}_k^+ &= \hat{x}_k^- + K_k \left(z_k - \hat{z}_k \right) \\ P_k^+ &= P_k^- - K_k P_k^- K_k^T \end{aligned} \tag{3.34}$$

This algorithm is initialized by

$$\begin{aligned} \hat{x}_0 &= E\left[x_0 \right] \\ \hat{x}_0^a &= \begin{bmatrix} \hat{x}_0^T & 0 & 0 \end{bmatrix}^T \\ P_0 &= E\left[\left(x_0 - \hat{x}_0 \right) \left(x_0 - \hat{x}_0 \right)^T \right] \\ P_0^a &= \begin{bmatrix} P_0 & 0 & 0 \\ 0 & Q & 0 \\ 0 & 0 & R \end{bmatrix} \end{aligned} \tag{3.35}$$

where the process and measurement noise are set to their mean value of zero and the process and measurement noise matrices augment the covariance matrix used to create the sigma point vectors.

The standard UKF algorithm for state estimation is summarized in Figure 10. In general this algorithm is $\mathrm{O}\left(L^3\right)$ computations which can be computationally intensive for systems with large state vectors.

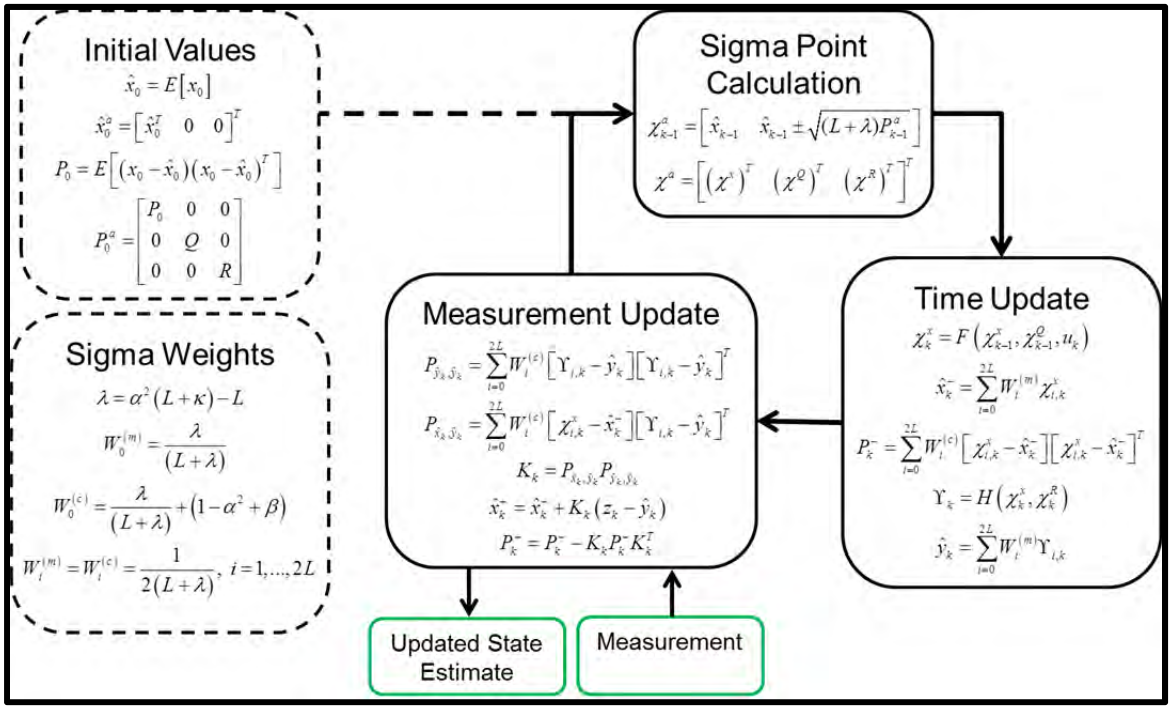

Figure 10. Standard UKF recursion.

2. Square Root UKF

For the standard UKF algorithm, the most computationally intensive operation is creating the sigma vectors at each time step, which requires calculating the matrix square root of the augmented error covariance matrix executed by the MATLAB *chol* function. The Cholesky factorization function *chol* uses $\mathrm{O}\left(L^3/6\right)$ computations and computes the matrix square root of the state covariance matrix given by $SS^T = P$. The square root UKF (SR-UKF) propagates S directly by using the linear algebra techniques: QR decomposition, Cholesky factorization updating, and efficient least square. QR decomposition is a technique used to calculate a matrix square root and requires only $\mathrm{O}\left(NL^2\right)$ computations, where N is the number of sigma vectors. The Cholesky factor

update corrects the output of the QR decomposition and requires only $\mathrm{O}\left(L^2\right)$ computations. In general, the SR-UKF computations are of the same order of magnitude as the standard UKF, but provide better numerical properties such as a guaranteed positive-semidefinite state covariance matrix [14]. This is an important quality as the numerical stability of the UKF depends on the state error covariance matrix remaining positive-semidefinite.

Using the same system described in Equation (3.23) and the augmented state vector in Equation (3.24), the augmented Cholesky factor is defined as

$$S_k^a = \begin{bmatrix} S_{k-1}^+ & 0 & 0 \\ 0 & \sqrt{Q} & 0 \\ 0 & 0 & \sqrt{R} \end{bmatrix} \tag{3.36}$$

The sigma point vectors are created with the same methods from the standard UKF with the exception that the matrix square root of the covariance is replaced by the Cholesky factor:

$$\begin{gathered} \chi_{k-1}^a = \begin{bmatrix} \hat{x}_{k-1} & \hat{x}_{k-1} + \sqrt{(L+\lambda)} S_k^a & \hat{x}_{k-1} - \sqrt{(L+\lambda)} S_k^a \end{bmatrix} \\ \chi^a = \begin{bmatrix} \left(\chi^x\right)^T & \left(\chi^Q\right)^T & \left(\chi^R\right)^T \end{bmatrix}^T \end{gathered} \tag{3.37}$$

The time update process is carried out in the same manner as the standard UKF with the exception that the state covariance estimate calculation step is replaced by the time update of the Cholesky factor S_k^- using the QR decomposition of the weighted sigma vectors and subsequent Cholesky factorization update or downdate depending on the value of the $W_0^{(c)}$ weight value. The QR decomposition is represented as the MATLAB operator *qr*, likewise the Cholesky update is represented by the MATLAB operator *cholupdate*:

$$
\begin{aligned}
\chi_k^x &= f\left(\chi_{k-1}^x, \chi_{k-1}^Q, u_k\right) \\
\hat{x}_k^- &= \sum_{i=0}^{2L} W_i^{(m)} \chi_{i,k}^x \\
S_k^- &= qr\left\{\left[\sqrt{W_1^{(c)}}\left(\chi_{1:2L,k}^x - \hat{x}_k^-\right)\right]\right\} \\
S_k^- &= cholupdate\left\{S_k^-, \left(\chi_{0,k}^x - \hat{x}_k^-\right), sign\left(W_0^{(c)}\right)\right\}
\end{aligned}
\tag{3.38}
$$

The measurement update equations are the same as the UKF, but replace the calculation of the measurement error covariance with the QR decomposition and Cholesky update:

$$
\begin{aligned}
\Upsilon_k &= h\left(\chi_k^x, \chi_k^R\right) \\
\hat{z}_k &= \sum_{i=0}^{2L} W_i^{(m)} \Upsilon_{i,k} \\
S_{\hat{z}_k} &= qr\left\{\left[\sqrt{W_1^{(c)}}\left(\Upsilon_{1:2L,k}^x - \hat{z}_k\right)\right]\right\} \\
S_{\hat{z}_k} &= cholupdate\left\{S_{\hat{z}_k}, \left(\Upsilon_{0,k} - \hat{z}_k\right), sign\left(W_0^{(c)}\right)\right\}
\end{aligned}
\tag{3.39}
$$

The error cross covariance calculation is the same as the UKF implementation. Efficient least squares is used to back-solve for the Kalman gain through the use of MATLAB's "/" operator:

$$
K_k = \left(P_{\hat{x}_k, \hat{z}_k} / S_{\hat{z}_k}^T\right) S_{\hat{z}_k} \tag{3.40}
$$

The measurement correction step is executed in the same manner as the UKF, but the correction to the error state is executed with a Choleskly downdate according to:

$$
\begin{aligned}
\hat{x}_k^+ &= \hat{x}_k^- + K_k\left(z_k - \hat{z}_k\right) \\
U &= K_k S_{\hat{z}_k} \\
S_k^+ &= cholupdate\left\{S_k^-, U, -1\right\}
\end{aligned}
\tag{3.41}
$$

The SR-UKF recursion is summarized in Figure 11 and initialized by

$$
\begin{aligned}
S_0 &= chol\left\{E\left[\left(x_0-\hat{x}_0\right)\left(x_0-\hat{x}_0\right)^T\right]\right\} \\
\hat{x}_0 &= E\left[x_0\right] \\
\hat{x}_0^a &= \begin{bmatrix}\hat{x}_0^T & 0 & 0\end{bmatrix}^T \\
S_0^a &= \begin{bmatrix} S_0 & 0 & 0 \\ 0 & \sqrt{Q} & 0 \\ 0 & 0 & \sqrt{R} \end{bmatrix}
\end{aligned}
\tag{3.42}
$$

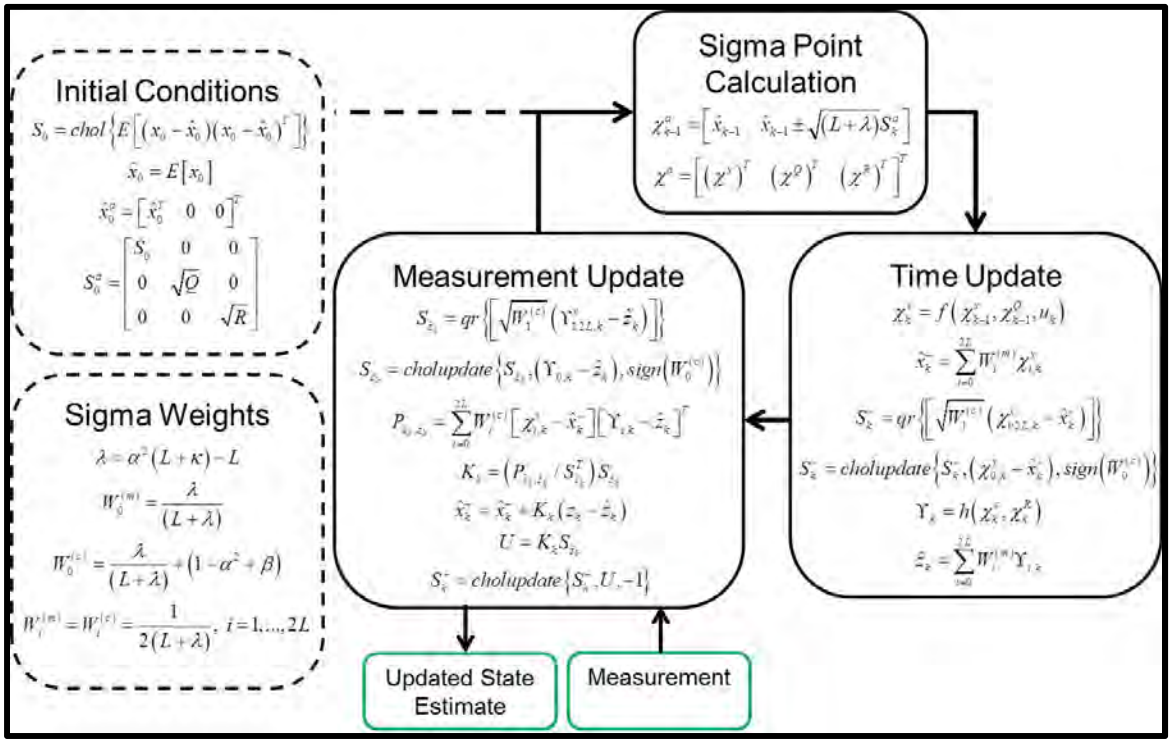

Figure 11. SR-UKF recursion.

3. Spherical Simplex

As both the standard UKF and SR-UKF are of the same computational complexity of $\mathrm{O}\left(L^3\right)$, one way to reduce the number of computations is to use less sigma point vectors to estimate the state mean and covariance. The spherical simplex method of selecting sigma points, developed by [15], requires only L+2 sigma vectors vice the 2L+1 required in both the UKF and SR-UKF by rearranging the sigma points on a hyper-sphere.

First, the zeroth weight is chosen such that $0 \le W_0 \le 1$. This weight only affects the fourth and higher order moments of the sigma point set. The remaining weights are calculated by

$$W_i = \frac{(1-W_0)}{L+1} \quad i = 1,...,L+1 \tag{3.43}$$

The weights can be scaled according to

$$w_i = \begin{cases} 1 + \dfrac{(W_0 - 1)}{\alpha^2} & \text{for } i = 0 \\ \dfrac{W_i}{\alpha^2} & \text{for } i \neq 0 \end{cases} \tag{3.44}$$

which helps to mitigate the effect of higher order terms.

Lastly, the zeroth weight of the scaled weight set can be modified to incorporate prior knowledge of the probability distribution by

$$\begin{aligned} W_0^{(c)} &= w_0 + \left(1 - \alpha^2 + \beta\right) \\ W_0^{(m)} &= w_0 \\ W_i^{(c)} &= W_i^{(m)} = w_i \end{aligned} \quad . \tag{3.45}$$

With the scaled spherical weights developed, the simplex matrix for selecting the sigma points is created using the algorithm

$$Z_0^1 = [0], Z_1^1 = \begin{bmatrix} \dfrac{-1}{\sqrt{2w_1}} \end{bmatrix}, Z_2^1 = \begin{bmatrix} \dfrac{1}{\sqrt{2w_1}} \end{bmatrix}$$

$$Z_i^j = \begin{cases} \begin{bmatrix} Z_0^{j-1} \\ 0 \end{bmatrix} & \text{for } i = 0 \\ \begin{bmatrix} Z_i^{j-1} \\ \dfrac{-1}{\sqrt{j(j+1)w_1}} \end{bmatrix} & \text{for } i = 1,..., j \\ \begin{bmatrix} 0_{j-1} \\ \dfrac{1}{\sqrt{j(j+1)w_1}} \end{bmatrix} & \text{for } i = j+1 \end{cases} \quad . \tag{3.46}$$

The spherical simplex sigma point vectors for the spherical simplex UKF (SSUKF) are created by

$$\chi_{k-1}^{a} = \hat{x}_{k-1}^{a} + Z_i \sqrt{P_{k-1}^{a}} \,. \tag{3.47}$$

The square root spherical simplex UKF (SR-SSUKF) sigma point vectors are created according to

$$\chi_{k-1}^{a} = \hat{x}_{k-1}^{a} + Z_i S_{k-1}^{a} \,. \tag{3.48}$$

The algorithms for the SSUKF and SR-SSUKF are identical to the standard UKF and SR-UKF, respectively, utilizing the spherical weights from Equation (3.45) and sigma point selection from Equations (3.47) or (3.48).

THIS PAGE INTENTINONALLY LEFT BLANK

IV. SIMULATION TOOLS AND ASSUMPTIONS

A. INTRODUCTION

In order to properly investigate the qualities of each navigation estimator, a controlled simulation environment was used to gather and analyze data. Utilizing a simulation environment required certain model and equation simplifications, but parameter values were chosen to mimic the true SEAFOX II data as close as possible with respect to sensor specifications.

B. SIMULATION ENVIRONMENT AND ASSUMPTIONS

1. Simulation Environment

Lacking a truth model for data recovered from the SEAFOX II sea trials, a robust simulation environment was used to evaluate and test each estimator. *Condor: The Competition Soaring Simulator* is a commercially available flight simulator that incorporates the use of a six degrees of freedom (6DOF) flight model and high fidelity physics engine that operates up to 500 cycles per second [17]. An application program interface (API) was recently developed that exports real-time flight telemetry data from Condor to MATLAB/Simulink and imports control surface commands from Simulink to Condor.

The following states of interest are available through the API at a frequency of 100 Hz:

$$X = \left[\phi, \lambda, h, x_n, y_e, z_d, v_n, v_e, v_d, \varphi, \theta, \psi, p, q, r, V_T\right] \tag{4.1}$$

- (ϕ, λ, h) are latitude, longitude, and altitude in degrees and meters, respectively.
- (x_n, y_e, z_d) are the LTP coordinates in meters.
- (v_n, v_e, v_d) are the LTP velocity components in m/s.
- (φ, θ, ψ) are the roll, pitch, and yaw Euler angles measured in radians.

- (p, q, r) are the roll, pitch, and yaw rates measured in radians per second.
- V_T is the true airspeed in m/s.

Due to software limitations in the API, the LTP frame acceleration states were not available during the development of this thesis.

2. Simulation Assumptions and Sensor Models

It is assumed that the Condor simulation environment uses a non-rotating flat earth model, therefore sidereal vector in the LTP frame $\omega_{ei}^{t} = 0$. Condor does not provide organic accelerometer measurements so these measurements were approximated following the derivation of specific force, F^b, from [18]. LTP acceleration was approximated using 3rd order filter shown in Figure 12, and the rotation matrix from LTP to body frame was formed with the available Euler angles using Equation (5.13):

$$F^b = R_t^b \left(\frac{d}{dt} V^t \right) + \omega_{bt}^b \times V^t - R_t^b g^t . \tag{4.2}$$

The Simulink representation of the filter used to estimate LTP acceleration is shown in Figure 12 and has the state space representation

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{x}_3 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ -k_1 & -k_2 & -k_3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ k_4 \end{bmatrix} u \tag{4.3}$$

where,

$$\begin{aligned} k_1 &= \omega_n^3 \\ k_2 &= 2\zeta\omega_n^2 \\ k_3 &= 2\zeta\omega_n \\ k_4 &= \omega_n^3 \end{aligned} \tag{4.4}$$

and ω_n is the natural frequency and ζ is a damping coefficient.

The filter bandwidth values were empirically determined to be $\omega_n = 10$ rad/s and $\zeta = 1.5$.

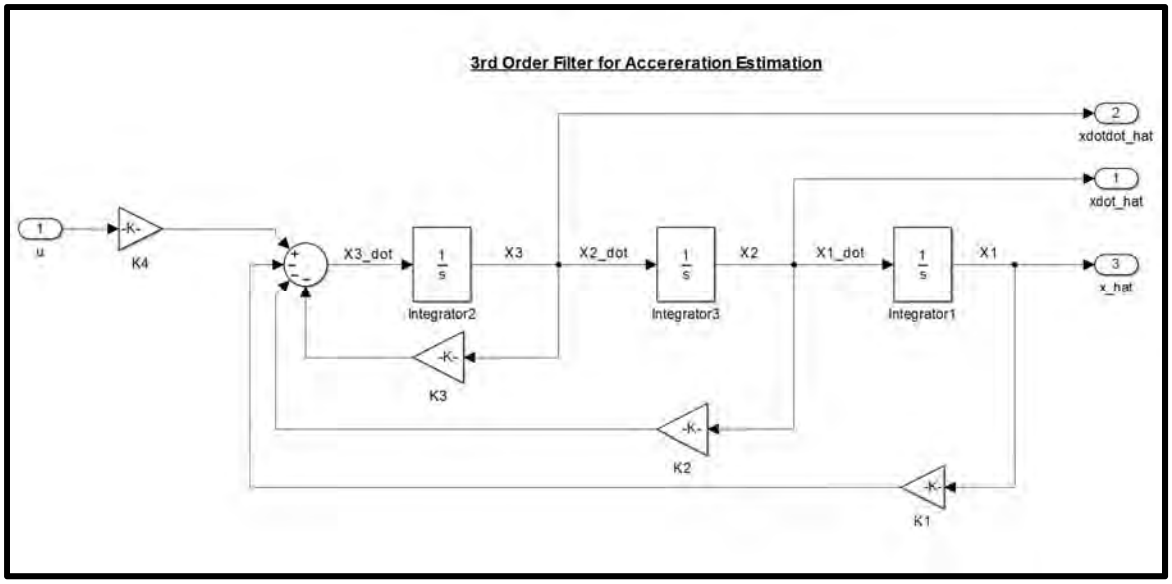

Figure 12. 3rd order filter for GPS acceleration estimation.

In an effort to make the simulated gyro and accelerometer measurements more realistic, normally distributed band-limited white noise and constant biases were added to each channel. The resulting gyro simulation sensor model used throughout the simulation trials was

$$\begin{aligned} \tilde{\omega}_{bi}^{b} &= \omega_{bi}^{b} + b_{gs} + v_{gs} \\ \dot{b}_{gs} &= 0 \\ v &\sim N\left(0, \sigma_{gs}^{2}\right) \end{aligned} \tag{4.5}$$

with associated accelerometer model:

$$\begin{aligned} \tilde{F}^{b} &= \left(R_{t}^{b}\dot{V}^{t} + \omega_{bt}^{b} \times V^{t} - R_{t}^{b} g^{t}\right) + b_{as} + v_{as} \\ \dot{b}_{as} &= 0 \\ v_{as} &\sim N\left(0, \sigma_{as}^{2}\right) \end{aligned} \quad . \tag{4.6}$$

The LTP position, speed, and heading from the output state vector were also corrupted with additive band-limited white noise resulting in the simulation models

$$\tilde{P}^{t} = P^{t} + v_{p}, \quad v_{p} \sim N\left(0, \sigma_{p}^{2}\right) \tag{4.7}$$

$$\tilde{V}_{T}^{t} = V_{T}^{t} + v_{T}, \quad v_{T} \sim N\left(0, \sigma_{T}^{2}\right) \tag{4.8}$$

$$\tilde{\psi} = \psi + v_{\psi}, \quad v_{\psi} \sim N\left(0, \sigma_{\psi}^2\right) \tag{4.9}$$

The power spectral density (PSD), defined as the variance of the noise, and bias values used for each simulated sensor throughout all the simulations are in Table 3. All values were chosen to closely mimic their counterpart onboard the SEAFOX II.

Sensor Noise and Bias Values for Condor Simulation		
Sensor	**PSD**	**Bias**
Gyro	$\sigma_{gs}^2 = \left(0.01 \text{ rad/s}\right)^2$	$b_{gs} = \begin{bmatrix} 2.4\times10^{-4} & -1\times10^{-4} & 2\times10^{-4} \end{bmatrix}^T$
Accelerometer	$\sigma_{as}^2 = \left(0.1 \text{ m/s}^2\right)^2$	$b_{as} = \begin{bmatrix} 1.5\times10^{-2} & -1.5\times10^{-2} & 1\times10^{-2} \end{bmatrix}^T$
GPS	$\sigma_p^2 = \left(3.2 \text{ m}\right)^2$	0
GPS Speed	$\sigma_T^2 = \left(.44 \text{ m/s}^2\right)^2$	0
GPS Heading	$\sigma_{\psi}^2 = \left(0.001 \text{ rad}\right)^2$	0

Table 3. Values for simulated sensor PSD and bias.

V. REFERENCE FRAMES

A. OVERVIEW

The design of a navigation system requires the transformation of sensor data between various frames to a common frame of reference. The accelerometer, gyroscope, and GPS measurement data must be transformed from their respective measurement frames of reference to a common frame prior to integration. Four frames of reference and the methods to transfer between systems are discussed in this chapter and used in the subsequent navigation system design: Earth centered inertial (ECI), Earth centered Earth fixed (ECEF), geographic (or navigation), and body frame.

B. REFERENCE FRAME DEFINITIONS

This section covers a brief overview of the four reference frames used throughout this thesis. All reference frame equations, transformations, and descriptions covered in this chapter are derived from chapter 2 of [8], which provides a thorough analysis of the topic.

1. Earth Centered Inertial

In the inertial frame of reference, Newton's second law of motion applies. The point of origin for an inertial frame of reference is arbitrary, but the three coordinate axes must be orthogonal to one another. Inertial sensors, such as an accelerometer and gyroscope, produce measurements relative to an inertial reference frame, resolved along the sensor's measurement axis.

The ECI reference frame is a non-rotating frame with its origin located at the Earth's center of mass. The z-axis is defined as parallel to the Earth's rotation axis. The x-axis is perpendicular to the z-axis, intercepting the sphere of the earth at 0^o latitude. The y-axis is orthogonal to both the x-axis and z-axis forming a right-handed coordinate system. See Figure 13 for further details.

2. Earth Centered Earth Fixed

The Earth centered Earth fixed (ECEF) reference frame is defined similarly to the ECI frame in that the origin is collocated at the center of Earth's mass and the z-axis is parallel to the Earth's rotation axis. The ECEF x-axis is fixed at the intersection of the Earth's sphere at 0^o latitude and 0^o longitude. The y-axis is perpendicular to both the z-axis and x-axis to complete a right-handed coordinate system. See Figure 13 for further detail. The ECEF frame rotates relative to the ECI frame with frequency:

$$\omega_{ie} \approx \left(\frac{1+365.25 \text{ cycle}}{(365.25 \times 24) \text{ hr}} \right) \left(\frac{2\pi \text{ rad/cycle}}{3600 \text{ sec/hr}} \right) = 7.292115 \times 10^{-5} \frac{\text{rad}}{\text{sec}} \tag{5.1}$$

3. Geographic (Navigation) Frame

The navigation frame is a local reference frame that moves with the vehicle, relative to the Earth's geoid. It is defined by the projection of the vehicle's origin onto the Earth's reference ellipsoid as shown in Figure 13. The z-axis points from the platform origin towards the origin of the reference ellipsoid. The x-axis points towards true north along the plane orthogonal to the z-axis. The y-axis points east, perpendicular to both the x-axis and z-axis, completing the right-handed coordinate system.

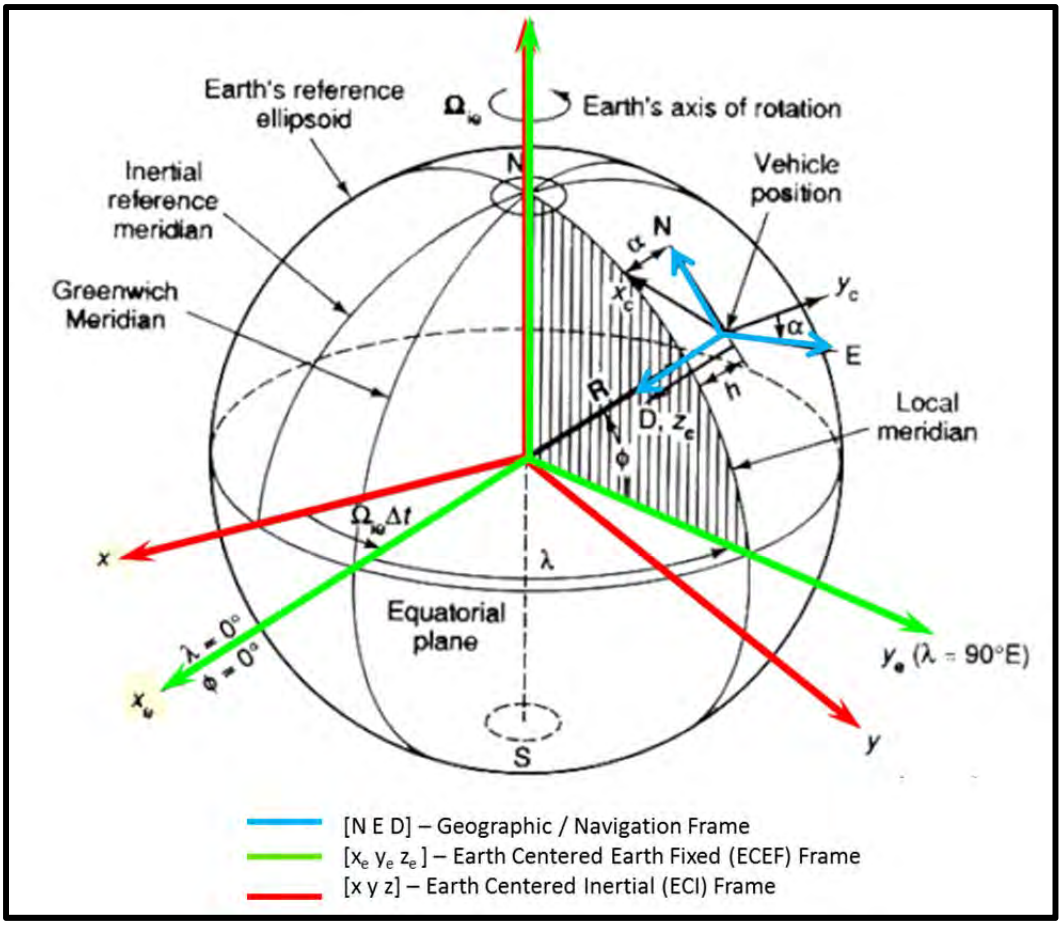

Figure 13. Navigation, ECEF, and ECI coordinate reference frames, after [19].

4. Body Frame

The body frame is rigidly attached to the vehicle. For this thesis, the body frame origin, vehicle center of gravity, and sensor origins are assumed to coincide which simplifies the transformations between sensor measurements and the frame the measurements are used in. The body frame x-axis, shown as the u-axis in Figure 14, points towards the front of the vehicle. The normal direction of vehicle motion is assumed to be in the x-direction. The z-axis is perpendicular to the x-axis and points at the bottom of the vehicle. The body y-axis is perpendicular to both the z-axis and y-axis to complete the right-hand coordinate system.

The rotation rate vector of the body frame relative to the inertial frame, resolved along the body axes is defined as $\omega_{bi}^{b} = [p, q, r]^{T}$, where p is the angular roll rate about

the x-axis, q is the angular pitch rate about the y-axis, and r is the angular yaw rate about the z-axis. Each angular rate is positive in the right-hand sense as shown in Figure 14.

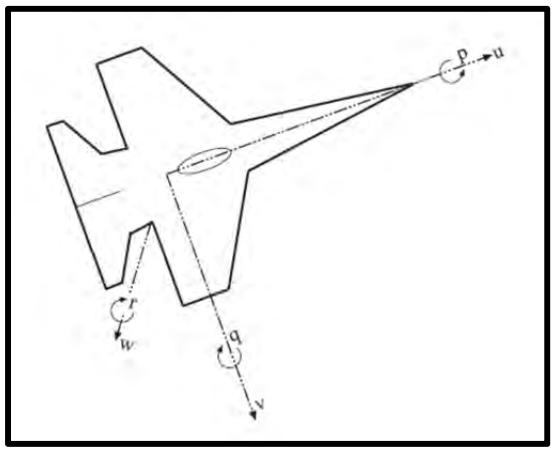

Figure 14. Body reference frame representation, from [8].

5. Local Tangent Plane

The local tangent plane (LTP) is a convenient frame of reference for navigation is local areas. The LTP frame uses a right hand north, east, down rectangular coordinate system where the origin is located in the vicinity of the vehicles operating area. For a stationary system located at the LTP fame origin, the LTP frame and navigation frame coincide.

C. EARTH REFERENCE ELLIPSOID

An accurate model of the Earth is vital to the accuracy of a navigation system. Figure 15 depicts a representation of the Earth's reference ellipsoid.

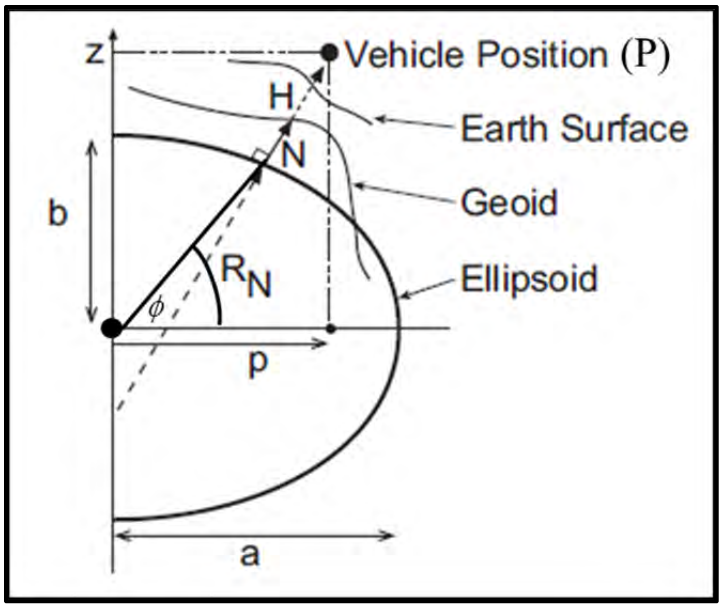

Figure 15. Earth reference ellipsoid, after [8].

The ellipsoid is defined by is semi-major axis (a) and its semi-minor axis (b). The axis values are defined by the WGS84 geodetic system and shown in Table 4.

WGS84 Parameters			
Parameter	**Symbol**	**Value**	**Units**
Semi-major Axis	a	6378137	Meters
Semi-minor Axis	b	6356752.314	Meters
Flatness	$f = \frac{a-b}{a}$	0.00335281	
Eccentricity	$e = \sqrt{f(2-f)}$	0.08181919	

Table 4. WGS84 parameter constants, from [8].

For a point P on the Earth's surface there is a north-south meridian plane of constant longitude that intercepts the Earth's ellipsoid. If arbitrary point P has a latitude of ϕ and longitude λ then the meridian radius can be described as:

$$R_M(\phi) = \frac{a\left(1-e^2\right)}{\left(1-e^2\sin^2(\phi)\right)^{\frac{3}{2}}} \tag{5.2}$$

For the same point *P*, the prime verticle, an east-west vertical plane that intercepts the reference ellipsoid as shown in Figure 15 that is normal to the meridian plane, has a radius of:

$$R_N(\phi) = \frac{a}{\sqrt{1 - e^2 \sin^2(\phi)}} \tag{5.3}$$

These radii are used for coordinate transformation between geodetic coordinates $[\phi, \lambda, h]^T$ and the ECEF coordinates $[x_e, y_e, z_e]^T$, where *h* is the altitude of *P* above the reference geoid.

D. ECEF TO NAVIGATION FRAME TRANSFORMATION

The navigation frame is a unique frame in that the origin of the navigation frame moves with the vehicle such that the vehicle's position expressed in the navigation frame is $X^n = [0, 0, -h]^T$. Latitude ϕ and longitude λ define the origin of the navigation frame on the reference ellipsoid. To obtain the earth relative velocity vector of the vehicle in the navigation frame, the ECEF velocity vector $V_e^e = \frac{d}{dt} X^e$ must be transformed to the navigation frame:

$$V_e^n = R_e^n V_e^e \tag{5.4}$$

The components of the velocity vector $V_e^n = [v_n, v_e, v_d]^T$ represent the instantaneous north, east, and down velocity components along the navigation frame axes. Since latitude, longitude, and altitude are the common values output by GPS receivers, the relationship between ECEF and geodetic position must be established and used to relate V_e^n to the rate of change of latitude, longitude, and altitude $(\dot{\phi}, \dot{\lambda}, \dot{h})$.

Using the radii from Equations (5.2) and (5.3), and the eccentricity constant from Table 3, the relationship between ECEF position and geodetic position is defined as:

$$\begin{aligned} x_e &= (R_N + h)\cos(\phi)\cos(\lambda) \\ y_e &= (R_N + h)\cos(\phi)\sin(\lambda) \\ z_e &= \left[R_N(1 - e^2) + h\right]\sin(\phi) \end{aligned} \tag{5.5}$$

The velocity vector V_e^e can be expressed as:

$$V_e^e = \frac{\partial X^e}{\partial \phi}\dot{\phi} + \frac{\partial X^e}{\partial \lambda}\dot{\lambda} + \frac{\partial X^e}{\partial h}\dot{h} \tag{5.6}$$

where the partial derivatives can be expressed as

$$\begin{aligned} \frac{\partial X^e}{\partial \phi} &= \left(R_M + h\right)\begin{bmatrix} -\sin(\phi)\cos(\lambda) \\ -\sin(\phi)\sin(\lambda) \\ \cos(\phi) \end{bmatrix}, \\ \frac{\partial X^e}{\partial \lambda} &= \left(R_N + h\right)\begin{bmatrix} -\cos(\phi)\sin(\lambda) \\ \cos(\phi)\cos(\lambda) \\ 0 \end{bmatrix}, \frac{\partial X^e}{\partial h} = \begin{bmatrix} \cos(\phi)\cos(\lambda) \\ \cos(\phi)\sin(\lambda) \\ \sin(\phi) \end{bmatrix} \end{aligned} \tag{5.7}$$

Equation (5.7) can then be rearranged to:

$$V_e^e = R_n^e \begin{bmatrix} \left(R_M + h\right) & 0 & 0 \\ 0 & \left(R_N + h\right)\cos(\phi) & 0 \\ 0 & 0 & -1 \end{bmatrix}\begin{bmatrix} \dot{\phi} \\ \dot{\lambda} \\ \dot{h} \end{bmatrix} \tag{5.8}$$

Finally, Equation (5.8) can be reduced to show the relation in Equation 4:

$$V_e^n = \begin{bmatrix} v_n \\ v_e \\ v_d \end{bmatrix} = \begin{bmatrix} \left(R_M + h\right)\dot{\phi} \\ \cos(\phi)\left(R_N + h\right)\dot{\lambda} \\ -\dot{h} \end{bmatrix} \tag{5.9}$$

In differential equation form, Equation (5.9) becomes:

$$\begin{bmatrix} \dot{\phi} \\ \dot{\lambda} \\ \dot{h} \end{bmatrix} = \begin{bmatrix} \dfrac{v_n}{\left(R_M + h\right)} \\ \dfrac{v_e}{\left(R_N + h\right)\cos(\phi)} \\ -v_d \end{bmatrix} \tag{5.10}$$

E. NAVIGATION TO BODY FRAME

The body frame is related to the navigation frame through the Euler angles (θ, φ, ψ) pitch, roll, and yaw, respectively. A direction cosine matrix (DCM) which is

used to transform vectors from the body to navigation frame can be formed by three plane rotations as shown in Figure 16.

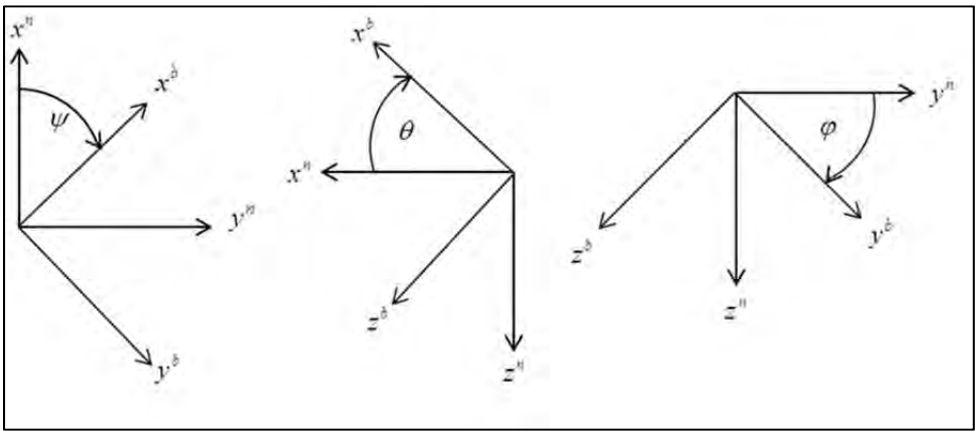

Figure 16. Plane rotations of yaw, pitch, and roll from navigation to body.

The three plane rotations from body to navigation frame are calculated by:

$$R_{z,\psi} = \begin{bmatrix} \cos(\psi) & \sin(\psi) & 0 \\ -\sin(\psi) & \cos(\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix}, R_{y,\theta} = \begin{bmatrix} \cos(\theta) & 0 & -\sin(\theta) \\ 0 & 1 & 0 \\ \sin(\theta) & 0 & \cos(\theta) \end{bmatrix},$$
$$R_{x,\varphi} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\varphi) & \sin(\varphi) \\ 0 & -\sin(\varphi) & \cos(\varphi) \end{bmatrix} \tag{5.11}$$

To create the DCM, R_n^b, the plane rotations from Equation (5.11) are multiplied in a 32–-1 sequence

$$R_n^b = R_{x,\varphi} R_{y,\theta} R_{z,\psi} \tag{5.12}$$

resulting in the rotation matrix:

$$R_n^b = \begin{bmatrix} \mathrm{c}\psi\,\mathrm{c}\theta & \mathrm{s}\psi\,\mathrm{c}\theta & -\mathrm{s}\theta \\ c\psi s\theta s\varphi - s\psi c\varphi & s\psi s\theta s\varphi + c\psi c\varphi & c\theta s\varphi \\ c\psi s\theta c\varphi + s\psi s\varphi & s\psi s\theta c\varphi - c\psi s\varphi & c\theta c\varphi \end{bmatrix} \tag{5.13}$$

where $s = \sin(\cdot)$, and $c = \cos(\cdot)$.

This rotation matrix has properties such that the rotation matrix that transforms vectors from the body to navigation frame is the transpose of the defined DCM:

$$R_b^n = \left(R_n^b\right)^T \tag{5.14}$$

The Euler angles (φ, θ, ψ) can be recovered from the DCM in Equation (5.13) via:

$$\begin{aligned} \varphi &= \tan^{-1}\left(\frac{R_n^b(2,3)}{R_n^b(3,3)}\right),\ \theta = -\sin^{-1}\left(R_n^b(1,3)\right), \\ \psi &= \tan^{-1}\left(\frac{R_n^b(1,2)}{R_n^b(1,1)}\right) \end{aligned} \tag{5.15}$$

where $R_n^b(row, colum)$ marks the appropriate matrix indices.

For local navigation in the vicinity of the local tangent plane origin, the rotation matrices and Euler angles can be assumed to be equal resulting in

$$\begin{aligned} R_b^t &= R_b^n \\ R_t^b &= \left(R_b^t\right)^T \end{aligned} \tag{5.16}$$

F. ECEF TO LOCAL TANGENT PLANE

A vehicle's arbitrary position in the ECEF frame is defined as $X_{vehicle}^{e} = \begin{bmatrix} x_e & y_e & z_e \end{bmatrix}^T$, and the origin of the LTP frame resolved in the ECEF frame is defined as $X_{LTP}^{e} = \begin{bmatrix} x_0 & y_0 & z_0 \end{bmatrix}^T$. The vector that connects the LTP origin to an arbitrary ECEF position is

$$\Delta X^e = X_{vehicle}^{e} - X_{LTP}^{e}. \tag{5.17}$$

Two plane rotations are required to transform a vector defined in ECEF to LTP. First the vector is rotated about the ECEF z-axis by the angle λ (longitude) then vertically by angle ϕ (latitude) to form the rotation matrix from ECEF to LTP as

$$R_e^t = \begin{bmatrix} c\lambda & s\lambda & 0 \\ -s\lambda & c\lambda & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} -s\phi & 0 & c\phi \\ 0 & 1 & 0 \\ c\phi & 0 & s\phi \end{bmatrix} = \begin{bmatrix} -s\phi c\lambda & -s\phi s\lambda & c\phi \\ -s\lambda & c\lambda & 0 \\ -c\phi c\lambda & -c\phi s\lambda & -s\phi \end{bmatrix}. \tag{5.18}$$

To resolve a vehicle's position in LTP, Equation (5.17) is rotated by the rotation matrix in Equation (5.18) resulting in

$$X^t = R_e^t \Delta X^e$$

$$\begin{bmatrix} x_n \\ y_e \\ z_d \end{bmatrix}^t = R_e^t \left(\begin{bmatrix} x_e \\ y_e \\ z_e \end{bmatrix}^e - \begin{bmatrix} x_0 \\ y_0 \\ z_0 \end{bmatrix}^e \right). \tag{5.19}$$

VI. NAVIGATION STATE ESTIMATOR DESIGN

A. OVERVIEW

A complimentary filter fuses the information from multiple sensors operating at various frequencies to produce a combined estimate of a desired state or parameter. In this implementation, a high rate sensor is used as the input to a system process and the output is corrected by a second sensor that operates at a much lower data output rate. For attitude estimation, the rate gyro measurements are integrated to estimate the vehicle's orientation. Depending on the quality of the gyro, these estimates can significantly drift. Likewise, the accelerometer measurements are integrated once for velocity estimates and a second time for position estimates which will lead to large dead reckoning errors without correction from an external source. Further discussion on complimentary filters can be found in chapter 7 of [8].

B. PROCESS MODEL

In a standard combined position, velocity, and attitude process model as described in chapter 11 of [8], the number of estimated states for even a simple navigation system can number around 15 when considering three dimensional position, velocity, and attitude, and gyro and accelerometer bias errors. The unscented Kalman filter process augments the already large state vector with process and measurement noise terms, growing the state vector to 30 states or more. When considering that the UKF must iterate $2L+1$ or $L+2$ times per time step depending on the sigma point selection method, where L is the number of states, it is clear that reducing the number of states is paramount to developing a practical real-time navigation estimator that features the UKF framework. To this end the process model was split into an attitude, heading, and reference system (AHRS) and an inertial navigation system (INS), reducing the number of states per estimator.

The cascaded estimator is broken into three major categories: sensor measurements, attitude determination, and position and velocity tracking. The cascaded process model used in each estimator is depicted in Figure 17. This cascaded design

closely follows the work presented in [20], [21], and [22]. The AHRS module receives inputs from the GPS, IMU, and previous time step estimates from the INS module and updates the vehicle's attitude. The updated attitude estimates are used to form a DCM from the body frame to the LTP reference frame, which is used by the INS module to update position and velocity estimates.

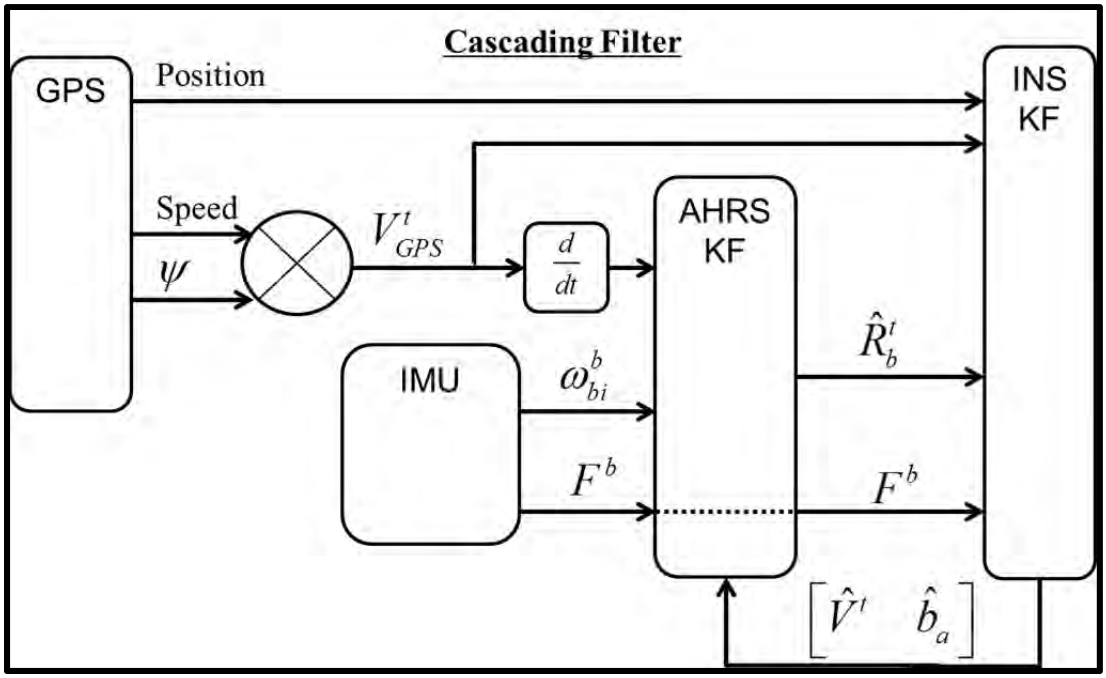

Figure 17. Cascaded Kalman filter for state estimation.

The state vector for the AHRS module estimators is:

$$\hat{X}_{AHRS} = \left[\hat{q}_0 \quad \hat{q}_1 \quad \hat{q}_2 \quad \hat{q}_3 \quad \hat{b}_{gs,p} \quad \hat{b}_{gs,q} \quad \hat{b}_{gs,r}\right]^T. \tag{6.1}$$

- $\left(\hat{q}_0 \quad \hat{q}_1 \quad \hat{q}_2 \quad \hat{q}_3\right)$ are the quaternion state estimates representing vessel orientation.
- $\left(\hat{b}_{gs,p} \quad \hat{b}_{gs,q} \quad \hat{b}_{gs,r}\right)$ are the gyro bias state estimates.

The state vector for the INS module estimator is:

$$\hat{X}_{INS} = \left[\hat{x}_n \quad \hat{y}_e \quad \hat{z}_d \quad \hat{V}_n \quad \hat{V}_e \quad \hat{V}_d \quad \hat{b}_{as,x} \quad \hat{b}_{as,y} \quad \hat{b}_{as,z}\right]^T. \tag{6.2}$$

- $\left(\hat{x}_n \quad \hat{y}_e \quad \hat{z}_d\right)$ are the north, east, and down LTP position estimates.
- $\left(\hat{V}_n \quad \hat{V}_e \quad \hat{V}_d\right)$ are the north, east, and down LTP velocity estimates.
- $\left(\hat{b}_{as,x} \quad \hat{b}_{as,y} \quad \hat{b}_{as,z}\right)$ are the body x, y, and z channel accelerometer bias estimates.

1. AHRS

The output of the IMU gyro is an angular rate measured in the body frame with respect to the inertial frame, represented in the body frame $\left(\omega_{bi}^{b}\right)$. The angular velocity of the body frame, with respect to the navigation frame, represented in the navigation frame $\left(\omega_{bn}^{b}\right)$ contains components of the sidereal rate $\left(\omega_{ei}^{n}\right)$ and transport rate $\left(\omega_{ne}^{n}\right)$ represented by the angular velocity of the navigation frame, with respect to the inertial frame, represented in the navigation frame $\left(\omega_{ni}^{n}\right)$:

$$\omega_{bn}^{b} = \omega_{bi}^{b} - R_{n}^{b}\omega_{ni}^{n} \tag{6.3}$$

$$\omega_{ni}^{n} = \omega_{ei}^{n} + \omega_{ne}^{n} = \begin{bmatrix} \omega_{ei}\cos(\phi) \\ 0 \\ -\omega_{ei}\sin(\phi) \end{bmatrix} + \begin{bmatrix} \dot{\lambda}\cos(\phi) \\ -\dot{\phi} \\ -\dot{\lambda}\sin(\phi) \end{bmatrix} = \begin{bmatrix} \left(\dot{\lambda}+\omega_{ei}\right)\cos(\phi) \\ -\dot{\phi} \\ -\left(\dot{\lambda}+\omega_{ei}\right)\sin(\phi) \end{bmatrix} \tag{6.4}$$

where ω_{ei}^{n} is the sidereal rate, defined as the angular velocity of the ECEF frame, with respect to the inertial frame, represented in the navigation frame at the latitude $\left(\phi\right)$ of the vehicle navigation frame origin, and ω_{ne}^{n} is the transport rate of the navigation frame origin in the ECEF frame resolved in navigation frame coordinates [8].

Equation (6.4) can be simplified by using a local tangent plane (LTP) assumption, denoted with the superscript *t*, which fixes the navigation frame to a constant location making $\omega_{ne}^{n} = 0$ resulting in

$$\omega_{bt}^{b} = \begin{bmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \end{bmatrix} = \omega_{bi}^{b} - R_{t}^{b}\omega_{ei}^{t} = \omega_{bi}^{b} - R_{t}^{b}\begin{bmatrix} \omega_{ei}\cos(\phi_0) \\ 0 \\ -\omega_{ei}\sin(\phi_0) \end{bmatrix}. \tag{6.5}$$

The standard representations of three-dimensional vehicle attitude are rotation matrices (or DCMs), Euler angles, and unit quaternions. Rotation matrix update equations are linear and lack singularities, but require nine differential equations. The Euler angle representation requires only three differential equations, but uses non-linear equations and contains a singularity when pitch passes 90 degrees. Unit quaternions use four differential equations, but the equations are linear and lack any singularities. Unit quaternions are used in the subsequent AHRS design due to the simplicity of their differential equations and ability to easily construct Euler angles and rotation matrices [23].

Following the quaternion derivations outlined in [23], the unit quaternion kinematic equation for attitude representation using the angular rate as described by Equation (6.5) is

$$\dot{q} = \frac{1}{2}\begin{bmatrix} 0 & \left(-\omega_{bn}^{b}\right)^{T} \\ \omega_{bn}^{b} & -S\left(\omega_{bn}^{b}\right) \end{bmatrix} q = \frac{1}{2}\begin{bmatrix} 0 & -\omega_1 & -\omega_2 & -\omega_3 \\ \omega_1 & 0 & \omega_3 & -\omega_2 \\ \omega_2 & -\omega_3 & 0 & \omega_1 \\ \omega_3 & \omega_2 & -\omega_1 & 0 \end{bmatrix} q \tag{6.6}$$

where $S(\bullet)$ is a skew symmetric operator.

A DCM from navigation or LTP to body frame can be formulated by using the relation

$$r^{n} = q \times r^{b} \times q^{*} = R_{b}^{n} r^{b} \tag{6.7}$$

where $r = \begin{bmatrix} r_x & r_y & r_z \end{bmatrix}^T$ is a three-dimensional vector, $q = \begin{bmatrix} q_0 & q_1 & q_2 & q_3 \end{bmatrix}^T$ is the unit quaternion, and $q^{*} = \begin{bmatrix} q_0 & -q_1 & -q_2 & -q_3 \end{bmatrix}^T$ is the unit quaternion complex conjugate.

The resulting DCM is

$$R_b^n = \begin{bmatrix} q_0^2 + q_1^2 - q_2^2 - q_3^2 & 2(q_1 q_2 - q_0 q_3) & 2(q_1 q_3 + q_0 q_2) \\ 2(q_1 q_2 + q_0 q_3) & q_0^2 - q_1^2 + q_2^2 - q_3^2 & 2(q_2 q_3 - q_0 q_1) \\ 2(q_1 q_3 - q_0 q_2) & 2(q_2 q_3 + q_0 q_1) & q_0^2 - q_1^2 - q_2^2 + q_3^2 \end{bmatrix} \tag{6.8}$$

$$R_n^b = \left(R_b^n\right)^T$$

The Euler angles are related to quaternions through the DCM as

$$\begin{aligned} \varphi &= \tan^{-1}\left(\frac{R_n^b(2,3)}{R_n^b(3,3)}\right) = \tan^{-1}\left(\frac{2(q_2 q_3 + q_0 q_1)}{q_0^2 - q_1^2 - q_2^2 + q_3^2}\right) \\ \theta &= \sin^{-1}\left(-R_n^b(1,3)\right) = \sin^{-1}\left(-2(q_1 q_3 - q_0 q_2)\right) \\ \psi &= \tan^{-1}\left(\frac{R_n^b(1,2)}{R_n^b(1,1)}\right) = \tan^{-1}\left(\frac{2(q_1 q_2 + q_0 q_3)}{q_0^2 + q_1^2 - q_2^2 - q_3^2}\right) \end{aligned} \tag{6.9}$$

In order to maintain accuracy, the unit quaternion must be normalized after each update by

$$q = \frac{q}{\sqrt{q^T q}} \tag{6.10}$$

to ensure that $\|q\| = 1$ at every time step.

2. INS Position and Velocity

In order to track the vessel's current position and velocity, an accelerometer is integrated through a series of equations that transform the measurements to the required frame of reference. Position and velocity in the navigation and LTP frames are derived here, as both frames were considered for implementation in the estimator design. The following equations for position and velocity are based on the derivations described in chapter 11 of [8].

As previously derived in Equation (5.10), the differential equation used to estimate the vehicle's position in the navigation reference frame is:

$$\begin{bmatrix} \dot{\phi} \\ \dot{\lambda} \\ \dot{h} \end{bmatrix} = \begin{bmatrix} \dfrac{v_n}{(R_M + h)} \\ \dfrac{v_e}{(R_N + h)\cos(\phi)} \\ -v_d \end{bmatrix} \tag{6.11}$$

The LTP frame origin is locked into a constant location, therefore position can be determined by directly integrating velocity:

$$\dot{P}^t = \begin{bmatrix} \dot{x}_n \\ \dot{y}_e \\ \dot{z}_d \end{bmatrix}^t = \begin{bmatrix} v_n \\ v_e \\ v_d \end{bmatrix}^t \tag{6.12}$$

To determine a vehicle velocity in the navigation or LTP frame it must first be resolved in the ECEF reference frame:

$$\dot{V}^e = R_b^e F^b + g^e - 2\left(\omega_{ei}^e \times V^e\right) \tag{6.13}$$

where ω_{ei}^e is the sidereal rate of the ECEF frame with respect to the inertial frame, represented in the ECEF frame, and F^b is the specific force as measured by an accelerometer. Specific force (or g-force) is a non-gravigational force per unit mass with units of acceleration (m/s^2) that contains linear, centripetal, and gravitational components of acceleration.

Since the navigation frame origin moves with the vehicle, the rate of change of the navigation frame with respect to the ECEF frame represents the vehicle velocity and is described by the differential equation

$$\dot{V}_e^n = R_e^n \left(\left(\omega_{en}^e \times V^e\right) + \dot{V}^e\right). \tag{6.14}$$

Substituting Equation (6.13) for $\dot{V}^e$ results in

$$\dot{V}_e^n = R_b^n F^b + g^n - \left(\omega_{ne}^n + 2\omega_{ei}^n\right) \times V^n \tag{6.15}$$

where ω_{ne}^n is the transport rate of the navigation frame with respect to the ECEF frame.

In the LTP frame, the rate of change of the tangent plane velocity with respect to the ECEF frame can be described by

$$\dot{V}_e^t = R\left(\left(\omega_{te}^e \times V^e\right) + \dot{V}^e\right) \tag{6.16}$$

where $\omega_{te}^e = 0$ since the LTP origin is in a fixed location.

Substituting Equation (6.13) for $\dot{V}^e$ reduces Equation (6.16) to

$$\dot{V}_e^t = R_b^t F^b + g^t - 2\left(\omega_{ei}^t \times V^t\right). \tag{6.17}$$

The LTP reference frame was selected as the desired design reference frame onboard the SEAFOX II as the boat normally operates in local waters and does not travel significant distances in the course of a single mission.

3. SEAFOX II IMU Model

The accelerometer and gyroscope in the IMU are not ideal sensors; they contain multiple sources of error and noise. Identifying the characteristics of the noise and error sources is a critical component of an accurate and robust navigation system. Table 2 in Chapter II lists the manufacturer's analysis of the IMU noise and error statistics. By sampling each IMU channel over a short time interval while the vehicle is stationary, the lower bound of the noise statistics can be verified. Figures 18 and 19 are representative measurements from each IMU channel over a 10 second sampling period, sampled at 100 Hz data output rate while the vehicle is stationary.

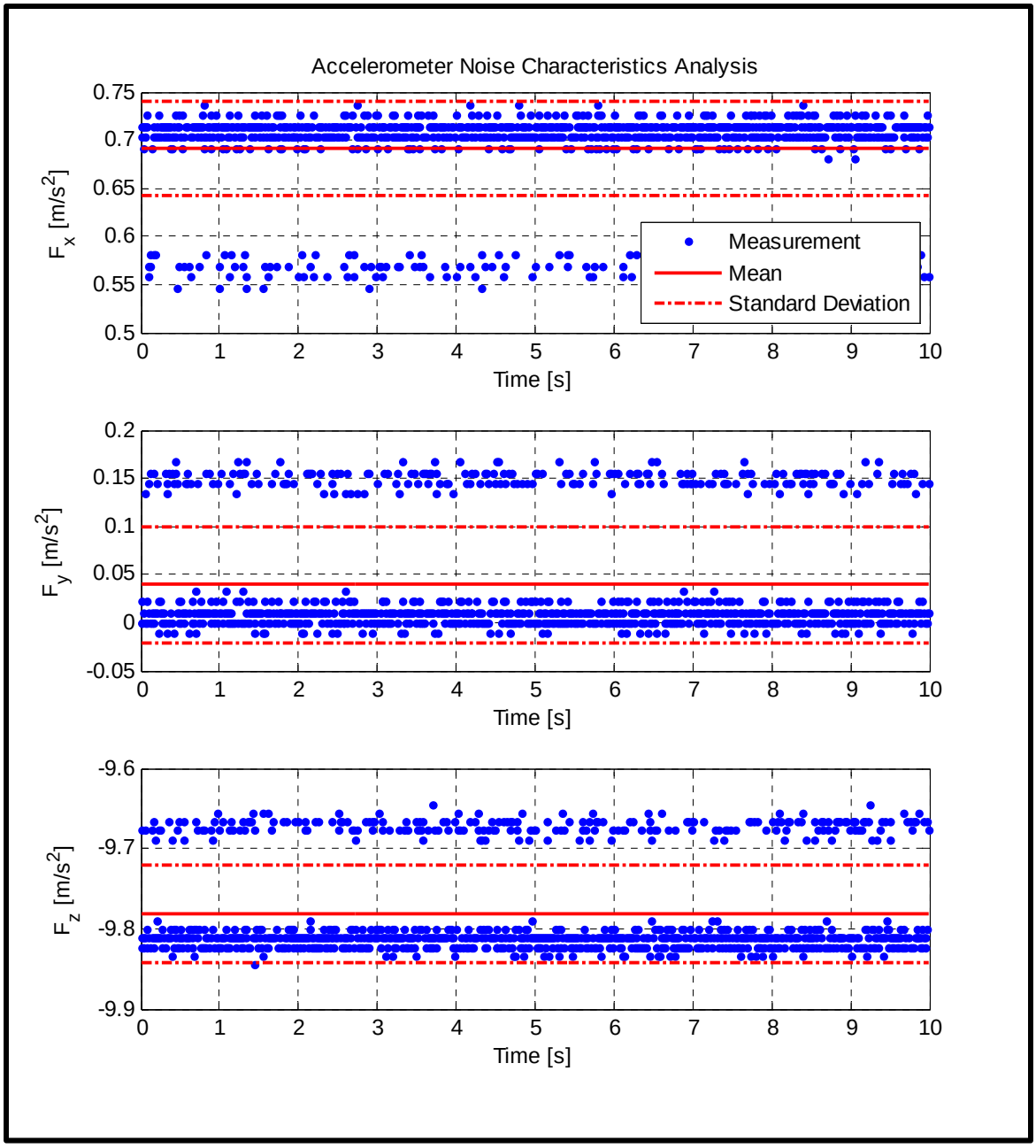

Figure 18. Measurements from the IMU accelerometer channels.

The standard deviation bounds shown in Figures 18 and 19 are the standard deviations of the data set shown, not of the manufacturer's specifications. Using MATLAB's standard deviation command *std* to analyze the data, the standard deviation for each accelerometer channel while stationary is $\sigma_a = \begin{bmatrix} 0.05 & 0.06 & 0.06 \end{bmatrix}$ m/s^2 , which corresponds to approximately 5 milli-g's, a common unit for specific force measurements. These standard deviation values are well within the manufacture's 100 milli-g noise maximum. All of the sampled data points fell within three standard deviations of the mean.

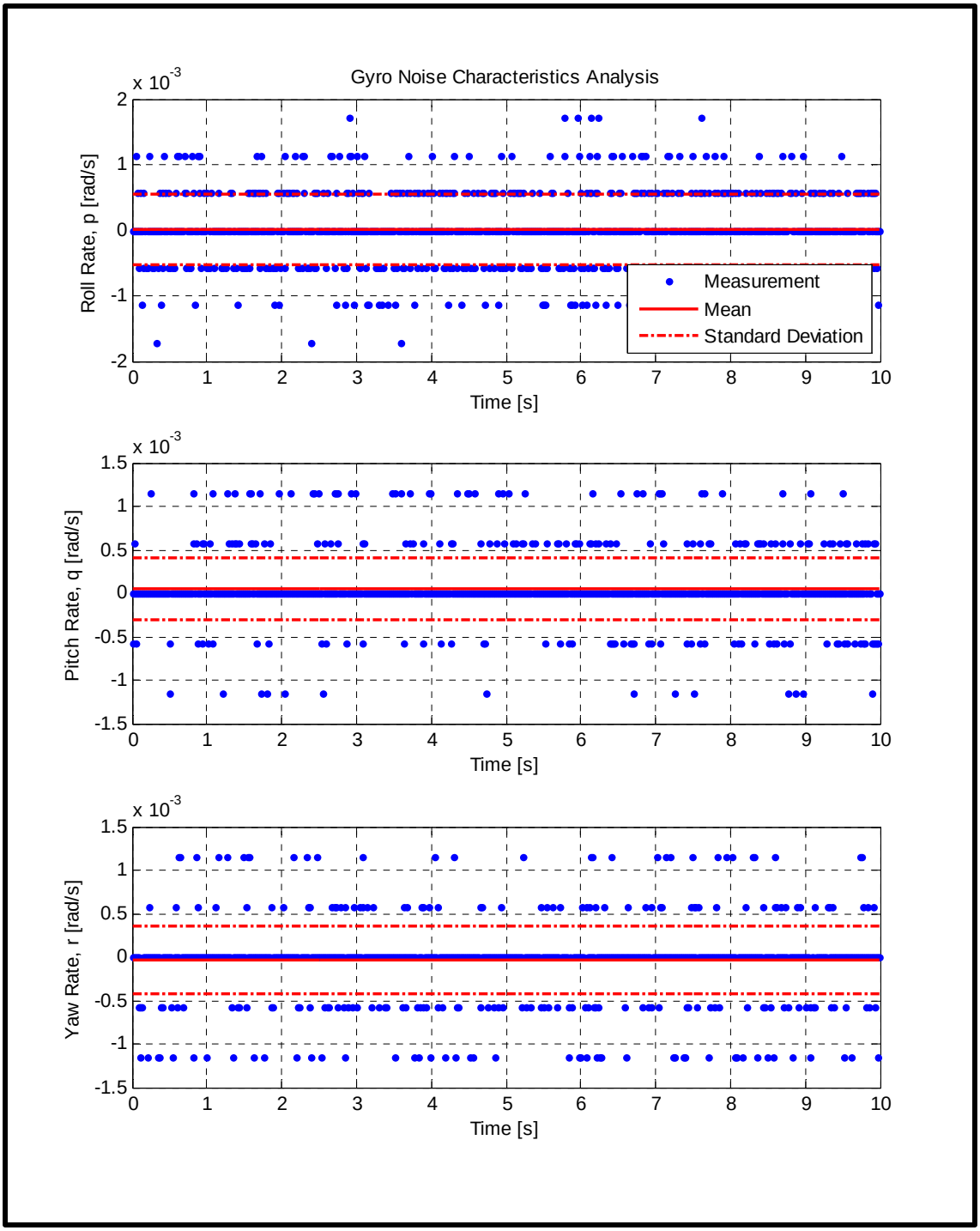

Figure 19. Measurements from the IMU gyro roll rate, pitch rate, and yaw rate channels.

Gyro measurements were acquired at the same time as the accelerometer measurements. The gyro measurements are much better distributed around the mean than the accelerometer measurements. Again, using MATLAB to analyze the measurements the standard deviation in the three gyro channels are $\sigma_g = \begin{bmatrix} 5\times10^{-4} & 3.5\times10^{-4} & 4\times10^{-4} \end{bmatrix}$ rad/s, all of which fall well below the manufacturer's maximum noise output. Since the vehicle was stationary during the sampling, the mean of each gyro channel serves as a good estimate for initializing the gyro bias values. The initial bias of the gyro is not always the same value when the gyro is initialized, so the bias must be measured at each restart of the IMU in order to initialize the estimator with accurate values.

In an effort to reduce the sensor model complexity, all noise sources are assumed to be zero mean Gaussian white noise. The error sources in each channel are assumed to be additive which means that they can be lumped into a single term. To account for noise sources that may vary with time, the derivative of the noise terms is assumed to be a Gaussian white noise term which will introduce a random walk to the bias estimates:

$$\begin{aligned} \tilde{F}^b &= a_{bi}^b - g^b + b_a + v_a \\ \dot{b}_a &= v_{b_a} \end{aligned} \tag{6.18}$$

$$\begin{aligned} \tilde{\omega}_{bi}^b &= \omega_{bi}^b + b_g + v_g \\ \dot{b}_g &= v_{b_g} \end{aligned} \tag{6.19}$$

The noise terms in Equation (6.18) and (6.19), $v_a \sim N\left(0, \sigma_a^2\right)$ and $v_g \sim N\left(0, \sigma_g^2\right)$, respectively, are assumed to be additive zero mean Gaussian white noise components of the accelerometer and gyro measurements. The random walk is described by the zero mean Gaussian white noise processes $v_{b_a} \sim N\left(0, \sigma_{b_a}^2\right)$ and $v_{b_g} \sim N\left(0, \sigma_{b_g}^2\right)$, where the σ_{b_a} and σ_{b_g} standard deviations are small tunable values.

C. MEASUREMENT MODELS

1. AHRS Measurement Model One

Accelerometers are commonly used as a means to correct drifting gyro measurements, as the accelerometer is able to provide noisy, yet accurate estimates of

pitch and roll via the measured gravity vector, as long as the vehicle is not accelerating. When a vehicle is accelerating, however, accelerometer measurements also include components of linear and centripetal acceleration that are not individually measurable. By making certain assumptions about the accelerometer measurements and introducing information from the GPS, better pitch and roll estimates can be calculated and used to correct the gyro measurements.

The first measurement model (MM1) investigated for this thesis estimates roll and pitch by manipulating the accelerometer measurement model as shown in Equation (4.6) to directly calculate roll and pitch angles. This approach is detailed in [20] and [21].

First, it is assumed that the Coriolis term is negligible, reducing the accelerometer model to

$$F^b = R_t^b \left(\frac{d}{dt} V^t \right) - R_t^b g^t . \tag{6.20}$$

Making the assumption that the GPS measurements are obtained in the inertial frame, the LTP accelerations are calculated by differentiating the GPS velocities. The north and east velocity components are calculated by

$$\begin{aligned} V_n^t &= V_T \cos(\psi) \\ V_e^t &= V_T \sin(\psi) \end{aligned} \tag{6.21}$$

where ψ is the GPS measured course over ground and V_T is the GPS measured speed of the vehicle.

The differentiation is carried out in simulation using the 3rd order filter described in Equation (4.3) and depicted in Figure 12 in Chapter IV. Defining the GPS acceleration vectors as $A_{GPS} = \begin{bmatrix} a_{GPS,x} & a_{GPS,y} & a_{GPS,z} \end{bmatrix}^T$, Equation (6.20) can be rewritten as

$$F^b = R_t^b \left(A_{GPS} - g^t \right). \tag{6.22}$$

The rotation matrix from LTP to body frame is then manipulated to define a new vector $r_{GPS} = \begin{bmatrix} r_x & r_y & r_z \end{bmatrix}^T$:

$$F^b = R_\varphi R_\theta R_\psi \left(A_{GPS} - g^t \right) = R_\varphi R_\theta \begin{bmatrix} a_{GPS,x}\cos(\psi) + a_{GPS,y}\sin(\psi) \\ -a_{GPS,x}\sin(\psi) + a_{GPS,y}\cos(\psi) \\ a_{GPS,z} - \|g\| \end{bmatrix}$$
$$F^b = \begin{bmatrix} c\theta & 0 & -s\theta \\ s\varphi s\theta & c\varphi & c\theta s\varphi \\ c\varphi s\theta & -s\varphi & c\varphi c\theta \end{bmatrix} \begin{bmatrix} r_x \\ r_y \\ r_z \end{bmatrix} \tag{6.23}$$

Since course over ground, ψ , is an output of the GPS, an analytical solution can be found to solve for pitch and roll. Pitch is first determined by manipulating

$$F_x^b = r_x \cos(\theta) - r_z \sin(\theta) \tag{6.24}$$

from the first row of Equation (6.23) to solve for θ, which after some manipulation yields

$$\theta = \tan^{-1}\left(\frac{-r_x r_z \pm F_x^b \sqrt{r_x^2 + r_z^2 - \left(F_x^b\right)^2}}{\left(F_x^b\right)^2 - r_z^2} \right) \tag{6.25}$$

Using this pitch value, roll can then be determined. The second row from Equation (6.23) can be rewritten to

$$F_y^b = r_x \sin(\theta)\sin(\varphi) + r_y \cos(\varphi) + r_z \cos(\theta)\sin(\varphi)$$
$$F_y^b = r_\theta \sin(\varphi) + r_y \cos(\varphi) \tag{6.26}$$

where

$$r_\theta = r_x \sin(\theta) + r_z \cos(\theta) \tag{6.27}$$

Solving Equation (6.26) for roll yields

$$\varphi = \tan^{-1}\left(\frac{r_\theta r_y \pm F_y^b \sqrt{r_y^2 + r_\theta^2 - \left(F_y^b\right)^2}}{\left(F_y^b\right)^2 - r_\theta^2} \right) \tag{6.28}$$

The measurement input vector for attitude estimation can now be formed as

$$z_{AHRS,1} = \begin{bmatrix} \varphi \\ \theta \\ \psi \end{bmatrix} = \begin{bmatrix} \tan^{-1}\left(\dfrac{r_\theta r_y - F_y^b \sqrt{r_y^2 + r_\theta^2 - \left(F_y^b\right)^2}}{\left(F_y^b\right)^2 - r_\theta^2} \right) \\ \tan^{-1}\left(\dfrac{-r_x r_z + F_x^b \sqrt{r_x^2 + r_z^2 - \left(F_x^b\right)^2}}{\left(F_x^b\right)^2 - r_z^2} \right) \\ \psi_{GPS} \end{bmatrix} \tag{6.29}$$

where the sign in front of the square root for pitch and roll is determined by the reference frame used. For the North-East-Down LTP frame, the signs are as shown.

This method provides good estimates of pitch and roll, but loses accuracy in roll estimation when the Coriolis term is large, which happens during sharp turns. If the assumption is made that angle of attack $\alpha = 0$ and sideslip $\beta = 0$ then it follows that the velocity in the body frame can be simplified to

$$V^b = \begin{bmatrix} u \\ v \\ w \end{bmatrix} = \begin{bmatrix} V_T \cos(\alpha)\sin(\beta) \\ V_T \sin(\beta) \\ V_T \sin(\alpha)\cos(\beta) \end{bmatrix} = \begin{bmatrix} V_T \\ 0 \\ 0 \end{bmatrix}. \tag{6.30}$$

Adding the simplified Coriolis force term back into the accelerometer measurement model yields:

$$F^b = R_t^b \left(\frac{d}{dt} V^t \right) + \left(\begin{bmatrix} p \\ q \\ r \end{bmatrix} \times \begin{bmatrix} V_T \\ 0 \\ 0 \end{bmatrix} \right) - R_t^b g^t. \tag{6.31}$$

The calculation for pitch angle remains the same as Equation (6.25), but for roll the calculation becomes:

$$\begin{aligned} F_y^b &= r_x \sin(\theta)\sin(\varphi) + r_y \cos(\varphi) + r_z \cos(\theta)\sin(\varphi) + V_T r \\ F_{c,y} &= \left(F_y^b - V_T r\right) = r_\theta \sin(\varphi) + r_y \cos(\varphi) \end{aligned} \tag{6.32}$$

where the Coriolis term is absorbed into the accelerometer term as $F_{c,y}$.

Finally, Equation (6.28) is adjusted by substituting in $F_{c,y}$ for F_y^b yielding:

$$\varphi = \tan^{-1}\left(\frac{r_\theta r_y \pm F_{c,y}\sqrt{r_y^2 + r_\theta^2 - \left(F_{c,y}\right)^2}}{\left(F_{c,y}\right)^2 - r_\theta^2}\right). \tag{6.33}$$

The new measurement vector for MM1 then becomes

$$z_{AHRS,1} = \begin{bmatrix}\varphi \\ \theta \\ \psi\end{bmatrix} = \begin{bmatrix} \tan^{-1}\left(\dfrac{r_\theta r_y + F_{c,y}\sqrt{r_y^2 + r_\theta^2 - \left(F_{c,y}\right)^2}}{\left(F_{c,y}\right)^2 - r_\theta^2}\right) \\ \tan^{-1}\left(\dfrac{-r_x r_z - F_x^b\sqrt{r_x^2 + r_z^2 - \left(F_x^b\right)^2}}{\left(F_x^b\right)^2 - r_z^2}\right) \\ \psi_{GPS} \end{bmatrix} \tag{6.34}$$

Figures 20 and 21 illustrate the accuracy of MM1 with the added Coriolis term as measured from Condor simulation data. Without the added term, the roll measurements overshoot, which requires judicious tuning of the noise matrix during tight turns. To avoid having to develop an adaptive tuning technique, the Coriolis term method is used in MM1 for subsequent estimator testing.

Figure 20 compares the true roll angle value from Condor with roll angles computed both with and without the Coriolis term added. Since the addition of the Coriolis term as previously defined has no effect on pitch, only one measurement is compared against Condor's true pitch angle values.

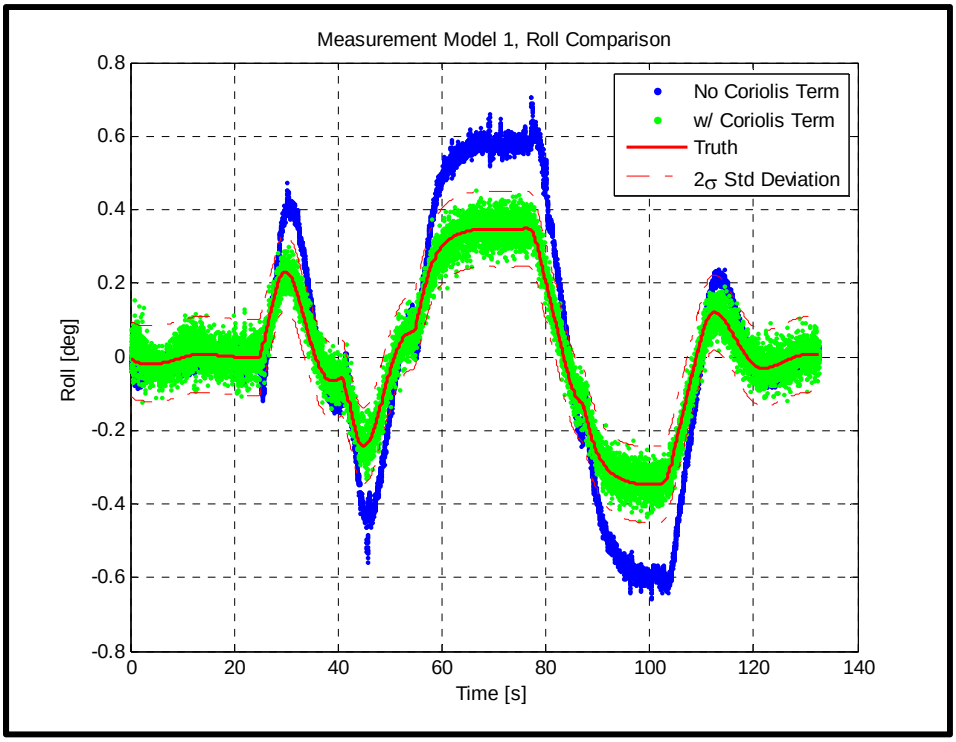

Figure 20. Roll measurements as approximate from MM1.

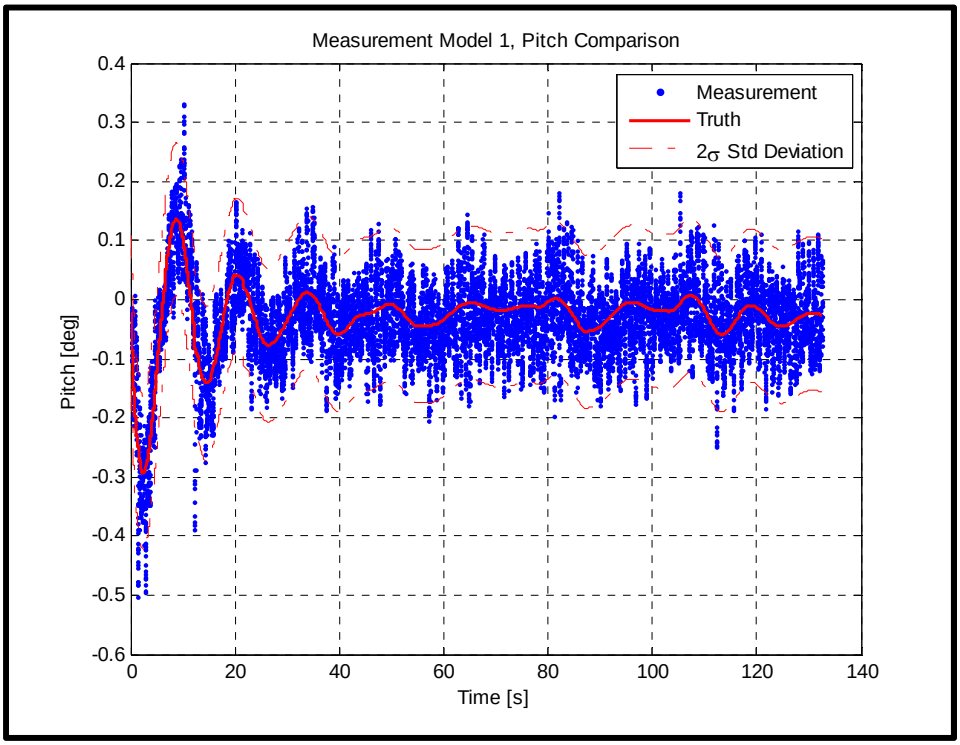

Figure 21. Pitch measurements as approximate from measurement model 1.

The standard deviation of the measured pitch and roll values was calculated by the general standard deviation equation:

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}\left(x_i - \mu\right)^2} \tag{6.35}$$

where the mean μ is assumed to be the true value of pitch or roll at each time step.

This yields pitch and roll standard deviations of $\sigma_\varphi = 0.03$ rad and $\sigma_\theta = 0.06$ rad. The 2σ values for pitch and roll are plotted in Figures 20 and 21 and show that the measurement estimates generally fall within that bound. This allows for pitch and roll sensor models to be approximated as the true values plus zero mean Gaussian white noise:

$$\begin{aligned} \tilde{\varphi} &= \varphi + v_\varphi, \quad v_\varphi \sim N\left(0, \sigma_\varphi^2\right) \\ \tilde{\theta} &= \theta + v_\theta, \quad v_\theta \sim N\left(0, \sigma_\theta^2\right) \end{aligned} \tag{6.36}$$

The measurement estimate that approximates roll, pitch, and yaw measurements based on the current state estimate is

$$\hat{z}_{AHRS,1} = \begin{bmatrix} \hat{\varphi} \\ \hat{\theta} \\ \hat{\psi} \end{bmatrix} = \begin{bmatrix} \tan^{-1}\left(\dfrac{2\left(\hat{q}_2\hat{q}_3 + \hat{q}_0\hat{q}_1\right)}{\hat{q}_0^2 - \hat{q}_1^2 - \hat{q}_2^2 + \hat{q}_3^2}\right) \\ \sin^{-1}\left(-2\left(\hat{q}_1\hat{q}_3 - \hat{q}_0\hat{q}_2\right)\right) \\ \tan^{-1}\left(\dfrac{2\left(\hat{q}_1\hat{q}_2 + \hat{q}_0\hat{q}_3\right)}{\hat{q}_0^2 + \hat{q}_1^2 - \hat{q}_2^2 - \hat{q}_3^2}\right) \end{bmatrix} + \begin{bmatrix} v_\varphi \\ v_\theta \\ v_\psi \end{bmatrix} \tag{6.37}$$

The measurement noise matrix for this measurement model is then defined as

$$R_{AHRS,1} = \begin{bmatrix} \sigma_\varphi^2 & 0 & 0 \\ 0 & \sigma_\theta^2 & 0 \\ 0 & 0 & \sigma_\psi^2 \end{bmatrix} \tag{6.38}$$

2. ARHS Measurement Model Two

Instead of trying to create a better external measurement to compliment the gyro, measurement model two (MM2) utilizes the available states to approximate the

accelerometer output. As in Equation (4.6), the accelerometer is assumed to have a sensor model of

$$\begin{aligned} \tilde{F}^b &= \left(R_t^b \dot{V}^t + \omega_{bt}^b \times R_t^b V^t - R_t^b g^t \right) + b_{as} + v_{as} \\ v_{as} &\sim N\left(0, \sigma_{as}^2\right) \end{aligned} . \tag{6.39}$$

The DCM from the LTP to body frame is constructed using the quaternions from the AHRS state vector. The LTP velocity and accelerometer bias terms are acquired from the INS module state vector. The angular rates and acceleration terms are treated as system inputs to the equations. The angular rates are measured from the gyro with current bias estimates and sidereal rates removed. The linear accelerations are created by differentiating the GPS velocities. Including the yaw estimate, the full measurement model yields:

$$\begin{aligned} \hat{z}_{AHRS,2} = \begin{bmatrix} \hat{F}^b \\ \hat{\psi} \end{bmatrix} &= \begin{bmatrix} \hat{R}_t^b \hat{A}_{GPS} + \left(\left(\omega_{bt}^b - \hat{b}_{gs} \right) \times \hat{R}_t^b \hat{V}^t \right) - \hat{R}_t^b g^t + \hat{b}_{as} \\ \tan^{-1}\left(\dfrac{2\left(\hat{q}_1 \hat{q}_2 + \hat{q}_0 \hat{q}_3 \right)}{\hat{q}_0^2 + \hat{q}_1^2 - \hat{q}_2^2 - \hat{q}_3^2} \right) \end{bmatrix} + \begin{bmatrix} v_{F^b} \\ v_\psi \end{bmatrix} \\ v_{F^b} &\sim N\left(0, \sigma_{F^b}^2\right) \\ v_\psi &\sim N\left(0, \sigma_\psi^2\right) \end{aligned} . \tag{6.40}$$

with associated measurement noise matrix

$$R_{2,AHRS} = \begin{bmatrix} \sigma_{F^b}^2 & 0 & 0 & 0 \\ 0 & \sigma_{F^b}^2 & 0 & 0 \\ 0 & 0 & \sigma_{F^b}^2 & 0 \\ 0 & 0 & 0 & \sigma_\psi^2 \end{bmatrix} \tag{6.41}$$

Figures 22–24 show accelerometer estimates generated using perfect state estimates, but noisy GPS acceleration and gyro measurements. All measurements were obtained via Condor simulation.

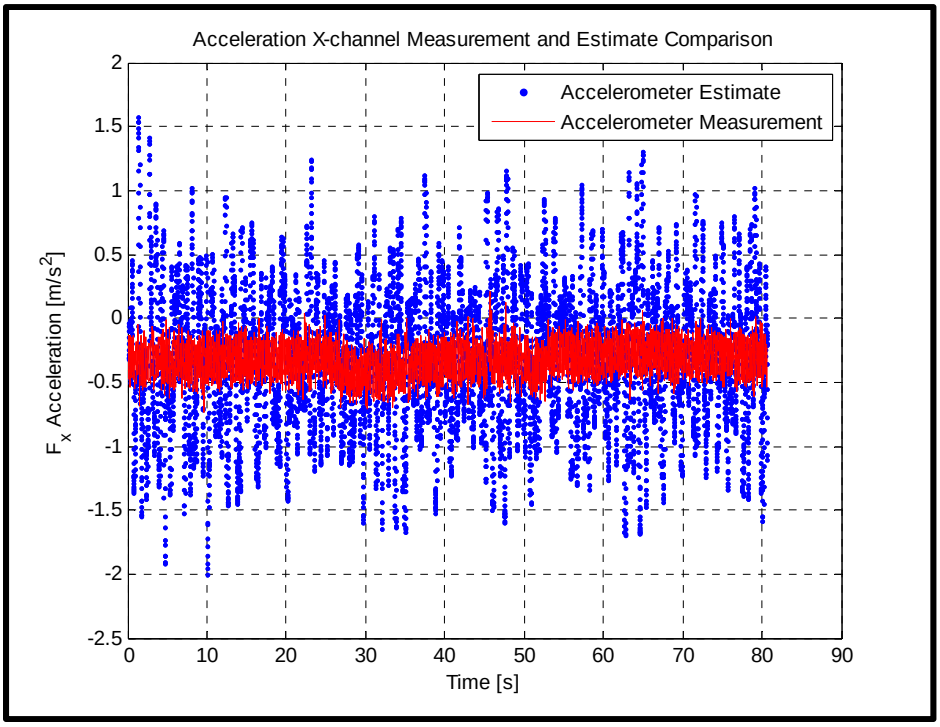

Figure 22. Acceleration X-channel comparison using the equations from MM2.

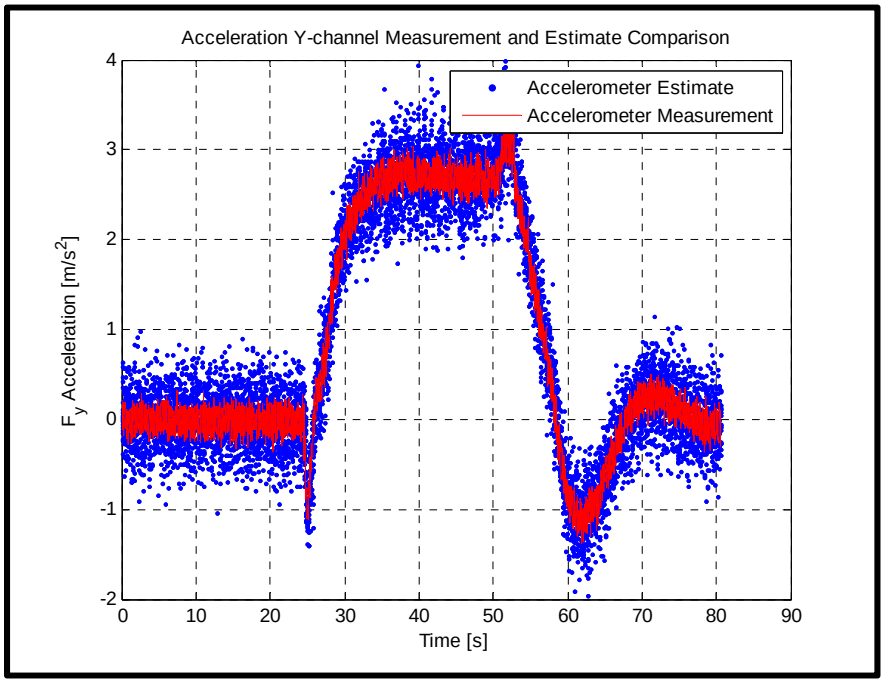

Figure 23. Acceleration Y-channel comparison using the equations from MM2.

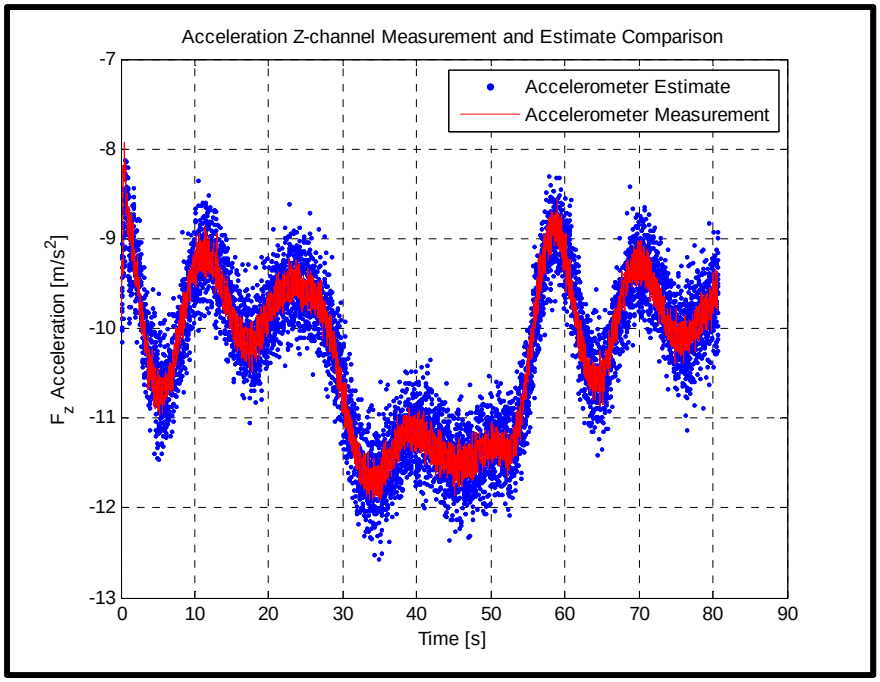

Figure 24. Acceleration Z-channel comparison using the equations from MM2.

It is clear from Figures 22–24 that the GPS acceleration estimates and gyro measurements contain more noise than the actual accelerometer measurements. Though noisy, the mean of the estimates contain the true value of the acceleration in each channel and are therefore a good representation of the accelerometer measurements.

3. INS Measurement Model

The measurement models for the position and velocity corrections are straightforward and do not require any unique mechanics. The estimates are pulled directly from the INS module state vector and compared to the GPS output. The GPS position and velocity measurements are assumed to be contaminated with additive zero mean white noise:

$$\hat{z}_{INS} = \begin{bmatrix} \hat{x}_n \\ \hat{y}_e \\ \hat{z}_d \\ \hat{V}_n \\ \hat{V}_e \\ \hat{V}_d \end{bmatrix} + \begin{bmatrix} v_{x_n} \\ v_{y_n} \\ v_{z_n} \\ v_{V_n} \\ v_{V_n} \\ v_{V_n} \end{bmatrix}. \tag{6.42}$$

The PSD for the noise values in Equation (6.42) are located in Chapter IV Table 4 for Condor simulated data and Table 5 for SEAFOX II data. These values are used to form the measurement noise matrix

$$R_{INS} = \begin{bmatrix} \sigma_p^2 & 0_{(3\times3)} \\ 0_{(3\times3)} & \sigma_V^2 \end{bmatrix} \tag{6.43}$$

ComNav Vector G1 Sensor Specifications	
Sensor	**Noise PSD**
GPS Position	$v_P \sim N\left(0,(5\text{ m})^2\right)$
GPS Velocity	$v_V \sim N\left(0,\left(0.5\text{ m/s}^2\right)^2\right)$

Table 5. Noise characteristics for the SEAFOX II GPS sensor, after [4].

D. EKF EQUATIONS

1. AHRS

As discussed in Chapter III, the extended Kalman filter requires first order derivative matrices to project the state, measurement, and covariance estimates. The AHRS process model is defined as

$$\dot{X}_{AHRS} = f_{AHRS}\left(X_{AHRS}, u, v\right) =$$
$$\begin{bmatrix} \dot{q} \\ \dot{b}_g \end{bmatrix} = \begin{bmatrix} \frac{1}{2}\begin{bmatrix} 0 & -\omega_1 & -\omega_2 & -\omega_3 \\ \omega_1 & 0 & \omega_3 & -\omega_2 \\ \omega_2 & -\omega_3 & 0 & \omega_1 \\ \omega_3 & \omega_2 & -\omega_1 & 0 \end{bmatrix} q \\ v_{b_g} \end{bmatrix} \tag{6.44}$$

where u is the process (gyro) measurement and v is the process noise.

According to the sensor model in Equation (6.19) the gyro measurement is:

$$\omega_{bt}^{b} = \begin{bmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \end{bmatrix} = \tilde{\omega}_{bi}^{b} - R_t^b \omega_{ei}^{t} - b_g - v_g \tag{6.45}$$

The state process Jacobian is formed by taking the partial derivative of the state differential equation with respect to each state, linearized around its current value, with noise terms equal to their expected value of zero:

$$F_{ARHS,k} = \left.\frac{\partial f_{AHRS}}{\partial X_{ARHS}}\right|_{\left(\hat{x}_k, u_k\right)}, \quad F_{ARHS,k} \in R^{7\times 7}$$
$$F_{ARHS,k} = \begin{bmatrix} \frac{\partial f_{ARHS}}{\partial q_0} & \frac{\partial f_{ARHS}}{\partial q_1} & \frac{\partial f_{ARHS}}{\partial q_2} & \frac{\partial f_{ARHS}}{\partial q_3} & \frac{\partial f_{ARHS}}{\partial b_g} \\ 0_{(3\times 1)} & 0_{(3\times 1)} & 0_{(3\times 1)} & 0_{(3\times 1)} & 0_{(3\times 3)} \end{bmatrix}. \tag{6.46}$$

The state excitation process Jacobian is formed by taking the partial derivative of the state process model with respect to the noise states, linearized around the current value of the state estimates:

$$G_{ARHS,k} = \left.\frac{\partial f_{ARHS}}{\partial v}\right|_{\left(\hat{X}_{ARHS,k}, u_k, v\right)}, \quad G_{ARHS,k} \in R^{7\times 6}$$
$$G_{ARHS,k} = \begin{bmatrix} \frac{\partial f_{ARHS}}{\partial v_g} & 0_{(3\times 3)} \\ 0_{(3\times 3)} & I_{(3\times 3)} \end{bmatrix}. \tag{6.47}$$

The state process and excitation Jacobian calculations are performed using continuous time equations. The EKF is run in discrete time, with discrete measurements,

therefore the state process and excitation matrices must be discretized. The following method for continuous state discretization is outlined in chapter 4 of [8] and utilizes matrix exponentials.

First, the matrix containing the process and excitation matrices is formed:

$$\Xi_k = \begin{bmatrix} -F_k & G_k Q_{AHRS} G_k^T \\ 0_{(7\times 7)} & F_k^T \end{bmatrix} T \tag{6.48}$$

where

$$Q_{ARHS} = \begin{bmatrix} \sigma_g^2 & 0_{(3\times 3)} \\ 0_{(3\times 3)} & \sigma_{b_g}^2 \end{bmatrix} \tag{6.49}$$

is the process noise matrix, and T is the filter time step length.

Next, the matrix exponential is performed using the MATLAB *expm* function:

$$\Upsilon = e^{\Xi} = \begin{bmatrix} -D & \Phi_k^{-1} Qd_k \\ 0_{(7\times 7)} & \Phi_k^T \end{bmatrix} \tag{6.50}$$

D is a dummy variable, Qd_k is the discrete time state excitation matrix, and Φ_k is the discrete time state transition matrix at time *k*.

The discrete time state excitation matrix and state transition matrix can be easily extracted from Equation (6.50) for use in the EKF error covariance time propagation equation:

$$P_{k+1}^{-} = \Phi_k P_k^{+} \Phi_k^T + Qd_k \tag{6.51}$$

Lastly, the measurement process Jacobians must be formed. The Jacobian of the first measurement process model, from Equation (6.37), is linearized with respect to the current estimates of the states, and given by

$$H_{AHRS,1,k} = \left. \frac{\partial z_{AHRS,1}}{\partial X_{ARHS}} \right|_{\left(\hat{X}_{ARHS,k},0\right)} = \begin{bmatrix} \frac{\partial z_{AHRS,1}}{\partial q_0} & \frac{\partial z_{AHRS,1}}{\partial q_1} & \frac{\partial z_{AHRS,1}}{\partial q_2} & \frac{\partial z_{AHRS,1}}{\partial q_3} & 0_{(3\times 3)} \end{bmatrix}. \tag{6.52}$$

The second measurement model process, given by Equation (6.40), is linearized to give:

$$H_{AHRS,2,k} \in R^{4\times 7}$$

$$H_{AHRS,2,k} = \left.\frac{\partial z_{AHRS,2}}{\partial X_{AHRS}}\right|_{\left(\hat{X}_{AHRS,k},0\right)} = \left[\begin{array}{ccccccc} \frac{\partial z_{AHRS,2}}{\partial q_0} & \frac{\partial z_{AHRS,2}}{\partial q_1} & \frac{\partial z_{AHRS,2}}{\partial q_2} & \frac{\partial z_{AHRS,2}}{\partial q_3} & \frac{\partial z_{AHRS,2}}{\partial b_{g,p}} & \frac{\partial z_{AHRS,2}}{\partial b_{g,q}} & \frac{\partial z_{AHRS,2}}{\partial b_{g,r}} \end{array}\right] . \quad (6.53)$$

The process, excitation, and measurement Jacobian matrices are extensive. The Appendix contains the MATLAB code for the EKF used in this analysis.

2. INS

The formation of the INS process model follows the same method used for developing the AHRS EKF equations. The state process model is defined from Equations (6.12), (6.17), and (6.18) as

$$\dot{X}_{INS} = f_{INS}\left(X_{INS}, u, v\right) =$$

$$\left[\begin{array}{c} \dot{P}^t \\ \dot{V}_e^t \\ \dot{b}_a \end{array}\right] = \left[\begin{array}{c} V^t \\ R\left(\tilde{F}^b\right) + g^t - 2S\left(\omega_{ei}^t\right)V^t \\ v_{b_a} \end{array}\right] \quad (6.54)$$

where

$$\tilde{F}^b = F^b + b_a + v_a . \quad (6.55)$$

The Jacobian of the state process model is then defined as

$$F_{INS,k} = \left.\frac{\partial f_{INS}}{\partial X_{INS}}\right|_{\left(\hat{X}_{INS,k},u_k\right)}, \quad F_{INS,k} \in R^{9\times 9}$$

$$F_{INS,k} = \left[\begin{array}{ccc} 0_{(3\times 3)} & I_{(3\times 3)} & 0_{(3\times 3)} \\ 0_{(3\times 3)} & S\left(\omega_{ei}^t\right) & -R_b^t \\ 0_{(3\times 3)} & 0_{(3\times 3)} & 0_{(3\times 3)} \end{array}\right] . \quad (6.56)$$

The Jacobian of the state excitation matrix is defined as

$$G_{INS,k} = \left.\frac{\partial f_{INS}}{\partial v}\right|_{\left(\hat{X}_{INS,k},u_k,v\right)}, \quad G_{INS,k} \in R^{9\times 6}$$

$$G_{INS,k} = \begin{bmatrix} 0_{(3\times3)} & 0_{(3\times3)} \\ -R_b^t & 0_{(3\times3)} \\ 0_{(3\times3)} & I_{(3\times3)} \end{bmatrix} . \tag{6.57}$$

The discrete time propagation matrices are formed using the same process from the AHRS estimator. The measurement process Jacobian is trivial since all the measurements are assumed to contain additive white noise:

$$H_{INS} \in R^{6\times9}$$

$$H_{INS} = \left.\frac{\partial z_{INS}}{\partial X_{INS}}\right|_{\left(\hat{X}_{INS,k},0\right)} = \begin{bmatrix} I_{(6\times6)} & 0_{(6\times3)} \end{bmatrix}. \tag{6.58}$$

E. UKF

Perhaps the largest benefit of using the unscented Kalman filter is the simplicity of the process time propagation and measurement updates. Instead of forming complex Jacobians to estimate the process and measurement models, the differential equations from Equations (6.44) and (6.54) are directly integrated using first order Euler integration methods. The augmented state vectors for the AHRS and INS modules accommodate the zero mean noise sources and are initialized according to Equation (3.35) as:

$$\hat{X}_{AHRS,1,0} = E\left[X_{AHRS,1,0}\right]$$
$$\hat{X}^a_{AHRS,1,0} = \begin{bmatrix} \hat{X}^T_{AHRS,1,0} & 0_{(1\times3)} & 0_{(1\times3)} \end{bmatrix}^T \tag{6.59}$$

$$\hat{X}_{AHRS,2,0} = E\left[X_{AHRS,2,0}\right]$$
$$\hat{X}^a_{AHRS,2,0} = \begin{bmatrix} \hat{X}^T_{AHRS,2,0} & 0_{(1\times3)} & 0_{(1\times4)} \end{bmatrix}^T \tag{6.60}$$

$$\hat{X}_{INS,0} = E\left[X_{INS,0}\right]$$
$$\hat{X}^a_{INS,0} = \begin{bmatrix} \hat{X}^T_{INS,0} & 0_{(1\times3)} & 0_{(1\times6)} \end{bmatrix}^T \tag{6.61}$$

The standard UKF and SR-UKF use $2L+1$ sigma vectors to calculate the process mean and covariance and the SSUKF and SR-SSUKF utilize $L+2$ as discussed in

Chapter III, where L is defined as the number of augmented states contained in the respective augmented state vector. Each variant of the UKF was built in accordance to the outline in Chapter III, the MATLAB code for which is available in the Appendix.

VII. DISCUSSION AND RESULTS

A. OVERVIEW

In order to determine the qualities of each Kalman filter covered in Chapter III, each estimator design was test with a common data set captured in the Condor simulation environment. Five Kalman filters and two attitude measurement models, equaling ten total navigation estimators were evaluated. Each estimator was initialized with identical state and covariance estimates, but with individually tuned measurement and process noise values that best enabled each filter to track the true value of the estimated states. First, each estimator was evaluated based on computation time of an identical data set to test for filter efficiency. To test for accuracy, each estimator was initialized with the true gyro and accelerometer bias values and evaluated on convergence to the true state values. Next, to test for robustness, each estimator was initialized with arbitrary gyro and accelerometer bias values and again evaluated on convergence to the true state values. Each estimator was then ranked by performance in a series of weighted categories to determine the best overall design. Lastly, the EKF and top ranked unscented Kalman filter were evaluated against SEAFOX II data.

B. FILTER TUNING

Perhaps the most time consuming portion of most Kalman filter design is properly tuning the measurement and noise matrices. Instead of iteratively evaluating entire data sets with constant test values for the noise and covariance matrices, each filter was tuned online by adjusting a series of gains while evaluating their real time effects on the state estimates in the Condor simulation environment. The PSD noise values used in the Condor simulations are summarized in Table 6. The AHRS process and measurement noise matrices for both MM1 and MM2 are initialized as:

$$
\begin{aligned}
Q_{AHRS} &= diag\left(\begin{bmatrix} \sigma_{gs}^2 & \sigma_{gs}^2 & \sigma_{gs}^2 \end{bmatrix}\right) \\
R_{AHRS,1} &= diag\left(\begin{bmatrix} k_1\sigma_{\varphi}^2 & k_2\sigma_{\theta}^2 & k_3\sigma_{\psi}^2 \end{bmatrix}\right) \\
R_{AHRS,2} &= diag\left(\begin{bmatrix} k_4\sigma_{F^b}^2 & k_5\sigma_{F^b}^2 & k_6\sigma_{F^b}^2 & k_7\sigma_{\psi}^2 \end{bmatrix}\right)
\end{aligned}
\tag{6.62}
$$

with INS estimator process and noise matrices initialized as

$$
\begin{aligned}
Q_{INS} &= diag\left(\begin{bmatrix} \sigma_{F^b}^2 & \sigma_{F^b}^2 & \sigma_{F^b}^2 \end{bmatrix}\right) \\
R_{INS,1} &= diag\left(\begin{bmatrix} k_8\sigma_P^2 & k_9\sigma_P^2 & k_{10}\sigma_P^2 & k_{11}\sigma_{V_n}^2 & k_{12}\sigma_{V_e}^2 & k_{13}\sigma_{V_d}^2 \end{bmatrix}\right) . \\
R_{INS,2} &= diag\left(\begin{bmatrix} k_{14}\sigma_P^2 & k_{15}\sigma_P^2 & k_{16}\sigma_P^2 & k_{17}\sigma_{V_n}^2 & k_{18}\sigma_{V_e}^2 & k_{19}\sigma_{V_d}^2 \end{bmatrix}\right)
\end{aligned}
\tag{6.63}
$$

Noise Characteristics for Simulation Sensor Outputs	
Sensors	**Noise PSD**
Gyro	$\sigma_{gs}^2 = \left(0.01 \text{ rad/s}\right)^2$
Accelerometer	$\sigma_{F^b}^2 = \left(0.1 \text{ rad/s}\right)^2$
Measured Euler Angles	$\sigma_{\varphi}^2 = \left(0.03 \text{ rad}\right)^2, \sigma_{\theta}^2 = \left(0.06 \text{ rad}\right)^2, \sigma_{\psi}^2 = \left(0.001 \text{ rad}\right)^2$
GPS Position	$\sigma_P^2 = \left(3 \text{ m}\right)^2$
GPS Velocity	$\sigma_{V_n}^2 = \left(0.3 \text{ m/s}^2\right)^2, \sigma_{V_e}^2 = \left(0.3 \text{ m/s}^2\right)^2, \sigma_{V_d}^2 = \left(0.001 \text{ m/s}^2\right)^2$

Table 6. Noise variance values for process and measurement noise in the Condor simulation environment.

All gain values in Equations (6.62) and (6.63) were scaled from a value of one to the final values listed in Table 7. The ratio of process noise to measurement noise determines the level of "trust" in the aiding sensor measurement by roughly scaling the Kalman gain according to

$$
K \propto \frac{Q}{R}. \tag{6.64}
$$

Large values for measurement noise result in a small Kalman gain allowing the process and process sensors to integrate with minimal correction. Small values for measurement noise increase the Kalman gain introducing large corrections to the process. Since the aided sensors are also corrupted by noise and error, the measurement gain had to be carefully tuned in order to avoid over correction and excess noise introduction to the system.

Process and Measurement Noise Scaling Factors						
	Gain	**EKF**	**UKF**	**SR-UKF**	**SSUKF**	**SR-SSUKF**
AHRS Measurement Model 1	k_1	1	0.05	0.025	0.05	1
	k_2	1	0.005	0.025	0.005	1
	k_3	1	1	1	1	1
	k_8	1	0.001	0.01	0.001	0.01
	k_9	1	0.001	0.01	0.001	0.01
	k_{10}	1	0.001	0.01	0.001	0.01
	k_{11}	1	0.005	0.005	0.005	0.01
	k_{12}	1	0.005	0.005	0.005	0.01
	k_{13}	1	1	1	1	1
AHRS Measurement Model 2	k_4	1	0.025	0.02	0.25	5
	k_5	1	0.025	0.05	0.25	1
	k_6	1	0.01	0.1	0.1	1
	k_7	1	1	1	1	1
	k_{14}	1	0.001	0.001	0.0001	0.001
	k_{15}	1	0.001	0.001	0.0001	0.001
	k_{16}	1	0.001	0.001	0.0001	0.001
	k_{17}	1	0.005	0.005	0.005	0.01
	k_{18}	1	0.005	0.005	0.005	0.01
	k_{19}	1	1	1	1	1

Table 7. Final process and measurement noise scaling factors.

Comparison of scaling factors associated with the EKF and UKF variants is difficult as the filter mechanics are different. The EKF Kalman gain is directly scaled by the measurement noise matrix whereas the UKF Kalman gain is indirectly scaled by the process and measurement noise matrices which are incorporated into the process of sigma point selection. With the exception of the SR-SSUKF, the remaining UKF variants are all scaled on or near the same order of magnitude.

It should be noted that the EKF designed with AHRS measurement model 2 is left out of all subsequent analysis as EKF(2) was completely unstable for all evaluated tuning gains. The instability was due to the high nonlinearity of the measurement model equations.

C. FILTER COMPUTATION SPEED COMPARISON

The largest detractor from the unscented Kalman filter and related filters (e.g. particle filter and Bayesian filter) is the number of iterations required to compute a single time step. In general, the number of UKF iterations per cycle is scaled based on the number of states being estimated. The standard UKF and SR-UKF use weights that require $2L+1$ iterations, where L is defined as the number of states plus the number of measurement noise terms and the number of process noise terms. The SSUKF and SR-SSUKF utilize the Spherical Simplex weight set which only requires L+2 iterations per cycle. The total iterations per cycle for each estimator module are listed in Table 8.

Module	UKF & SR-UKF	SSUKF & SR-SSUKF
AHRS (Model 1)	27	15
AHRS (Model 2)	29	16
INS	37	20

Table 8. Number of iterations per time step of each UKF variant for the given estimator design.

To evaluate the overall speed of each filter, an experiment was designed in Simulink to run a 10 second data set from Condor through each filter. The MATLAB functions *tic* and *toc* act as a high accuracy CPU stopwatch and were executed in the simulation start function and stop function callback, respectively, to measure the elapsed time for each filter. The results from this experiment are shown in Figure 25.

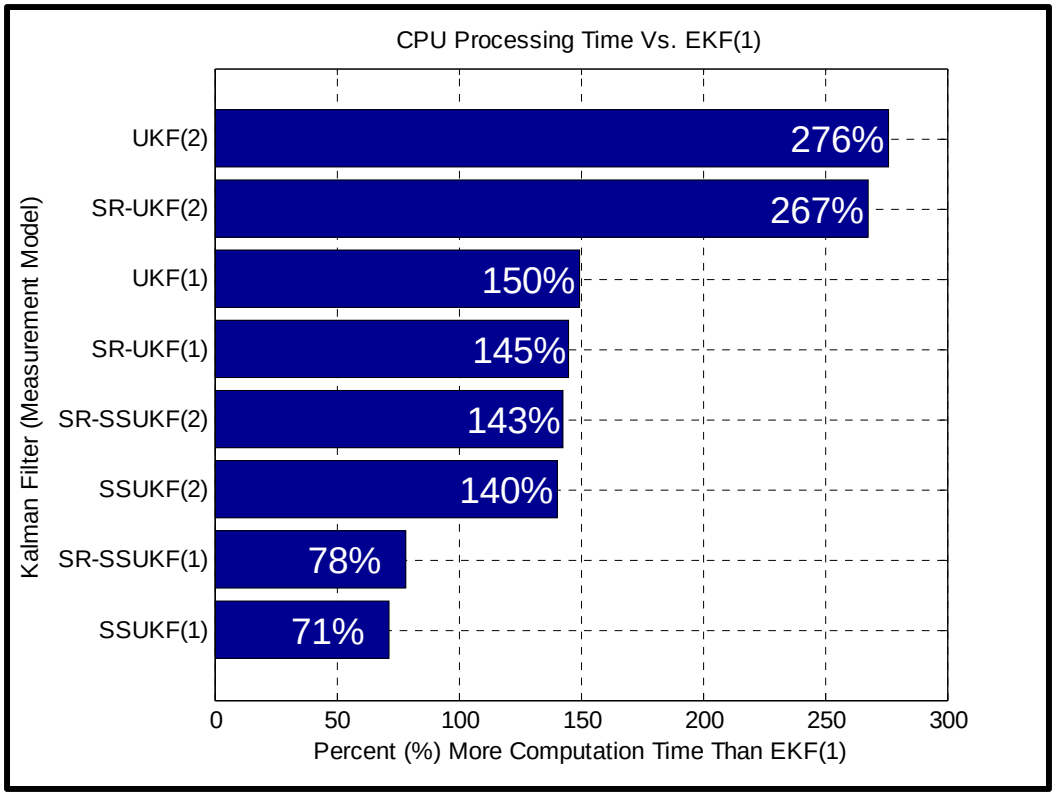

Figure 25. Comparison of relative computation time of a 10 seconds data set run in Simulink against the EKF(1) run time.

The results shown in Figure 25 are not surprising in that the unscented filters with spherical simplex weights outperformed those with standard weights as the spherical simplex method employs fewer sigma points and thus less iteration per time step. The SR-UKF and SR-SSUKF implementations were expected to slightly outperform their counterparts as the linear algebra operators that form the square root implementation are more efficient than the Cholesky square root operator. This result was true for the normal UKF sigma point scheme, but not so for the spherical simplex.

A second test for computational efficiency was run to minimize any computational overhead that may have come from using Simulink and its discrete time solver. Each estimator was built in the form of a MATLAB function file and was run in a script with a random vector input to measure the time of a single time step, again using the *tic* and *toc* MATLAB functions. Each estimator was run 25 times, the times of which were normalized against the EKF(1) run time and shown in Figure 26.

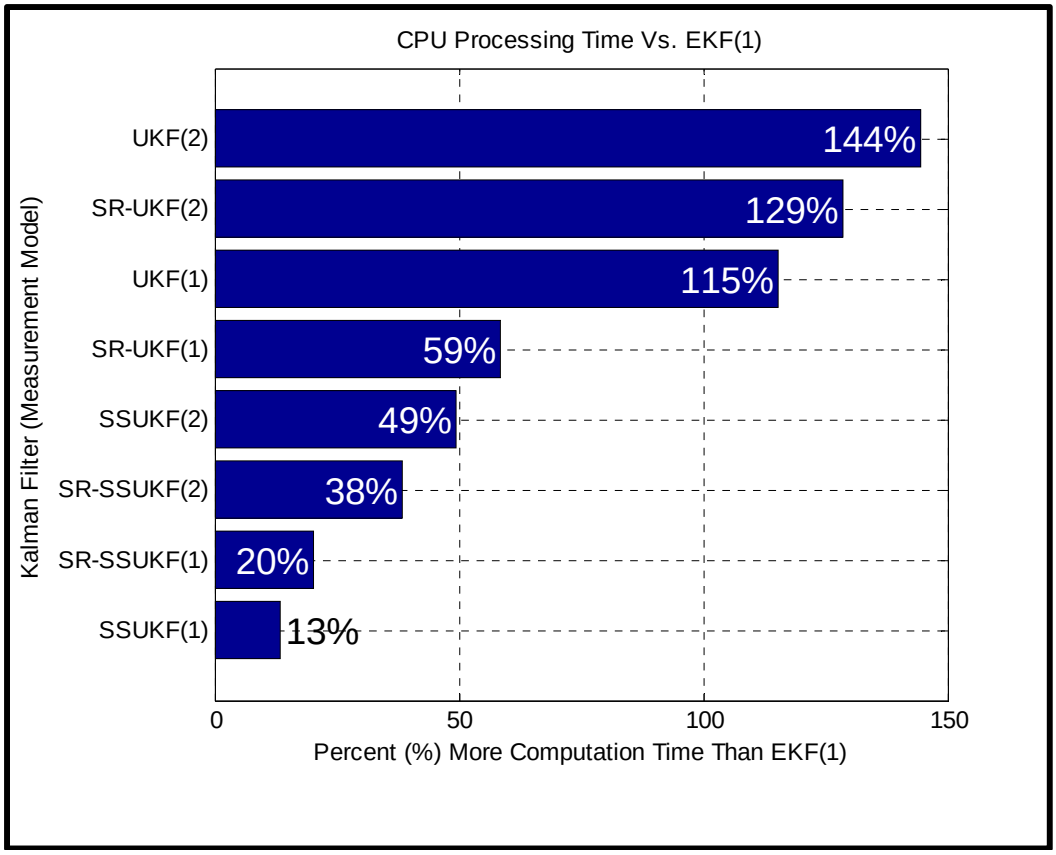

Figure 26. Comparison of relative computation time for a single iteration of each estimator function against EKF(1).

Running each estimator as a function call in MATLAB resulted in a significant increase in speed performance from the unscented filters. The largest performance increase came from UKF(2) and SR-UKF(2) which improved by 132% and 138%, respectively. The smallest improvement came from UKF(1) with a decrease of 35% computation time. While it is difficult to pinpoint what exactly differs between MATLAB and Simulink execution of certain function calls, what is apparent is that it is possible to reduce the relative computation time of an unscented Kalman filter to near EKF speed depending on the efficiency of the software used to execute the filter.

The MM2 estimators were expected to have increased computation times compared to their MM1 counterparts as there are slightly more sigma points to calculate and the formation of the measurement estimate $\hat{z}_{AHRS,2}$ from Equation (6.40) requires multiple rotation matrices and a cross product.

With the exception of the SR-SSUKF(1) estimator, the square root implementations outperformed their non-square root counterparts in Figure 26 by a wider margin than those in Figure 25. This may be due to differences between MATLAB and Simulink execution of the QR decomposition and Cholesky Update operators.

D. ATTITUDE PERFORMANCE

Two experiments were carried out to analyze the attitude performance of each estimator. For each experiment the filters were fed the same Condor data set and initialized with the same state and covariance estimates. For the first experiment, the gyro and accelerometer bias values were initialized to their true values. For the second experiment, the gyro and accelerometer values are initialized to zero and the covariance was adjusted to reduce trust in the bias channels, as shown in Table 9.

State and Covariance Initialization Values	
Initial Estimate	**AHRS Module Experiment 1**
Attitude	$\hat{q}(0) = [0.9529, 0.0055, -0.0172, 0.3029]$
Gyro Bias	$\hat{b}_g(0) = \left[2.4\times10^{-4}, -1.0\times10^{-4}, 2.0\times10^{-4}\right]$ rad/s
Covariance	$\hat{P}_{AHRS}(0) = diag\left(\left[10^{-5}, 10^{-5}, 10^{-5}, 10^{-5}, 10^{-6}, 10^{-6}, 10^{-6}\right]\right)$
Initial Estimate	**AHRS Module Experiment 2**
Attitude	$\hat{q}(0) = [0.9529, 0.0055, -0.0172, 0.3029]$
Gyro Bias	$\hat{b}_g(0) = [0, 0, 0]$ rad/s
Covariance	$\hat{P}_{AHRS}(0) = diag\left(\left[10^{-5}, 10^{-5}, 10^{-5}, 10^{-5}, 10^{-3}, 10^{-3}, 10^{-3}\right]\right)$
Initial Estimate	**INS Module Experiment 1**
Position	$\hat{X}^{NED}(0) = \left[3.8047\times10^{4}, -1.095\times10^{5}, 1.4180\times10^{3}\right]$ m
Velocity	$V^{NED}(0) = [25, 17, 0.6]$ m/s^2
Accelerometer Bias	$\hat{b}_a(0) = \left[1.5\times10^{-2}, -1.5\times10^{-2}, 1.0\times10^{-2}\right]$ m/s^2
Covariance	$\hat{P}_{INS}(0) = diag\left(\left[10^{-5}, 10^{-5}, 10^{-5}, 10^{-5}, 10^{-5}, 10^{-5}, 10^{-6}, 10^{-6}, 10^{-6}\right]\right)$
Initial Estimate	**INS Module Experiment 2**
Position	$\hat{X}^{NED}(0) = \left[3.8047\times10^{4}, -1.095\times10^{5}, 1.4180\times10^{3}\right]$ m
Velocity	$V^{NED}(0) = [25, 17, 0.6]$ m/s^2
Accelerometer Bias	$\hat{b}_a(0) = [0, 0, 0]$ m/s^2
Covariance	$\hat{P}_{INS}(0) = diag\left(\left[10^{-5}, 10^{-5}, 10^{-5}, 10^{-5}, 10^{-5}, 10^{-5}, 10^{-3}, 10^{-3}, 10^{-3}\right]\right)$

Table 9. Initial state and covariance estimates used initialize each navigation estimator.

As the SEAFOX II is natively able to produce a well filtered estimate of heading from its multiple GPS receivers, the main focus of this section will be to evaluate roll and pitch estimation. Figure 27 highlights each estimator's performance at the peak of a high rate turn from the first experiment's data set. The top graph shows MM1 estimators and the bottom section shows MM2 estimators.

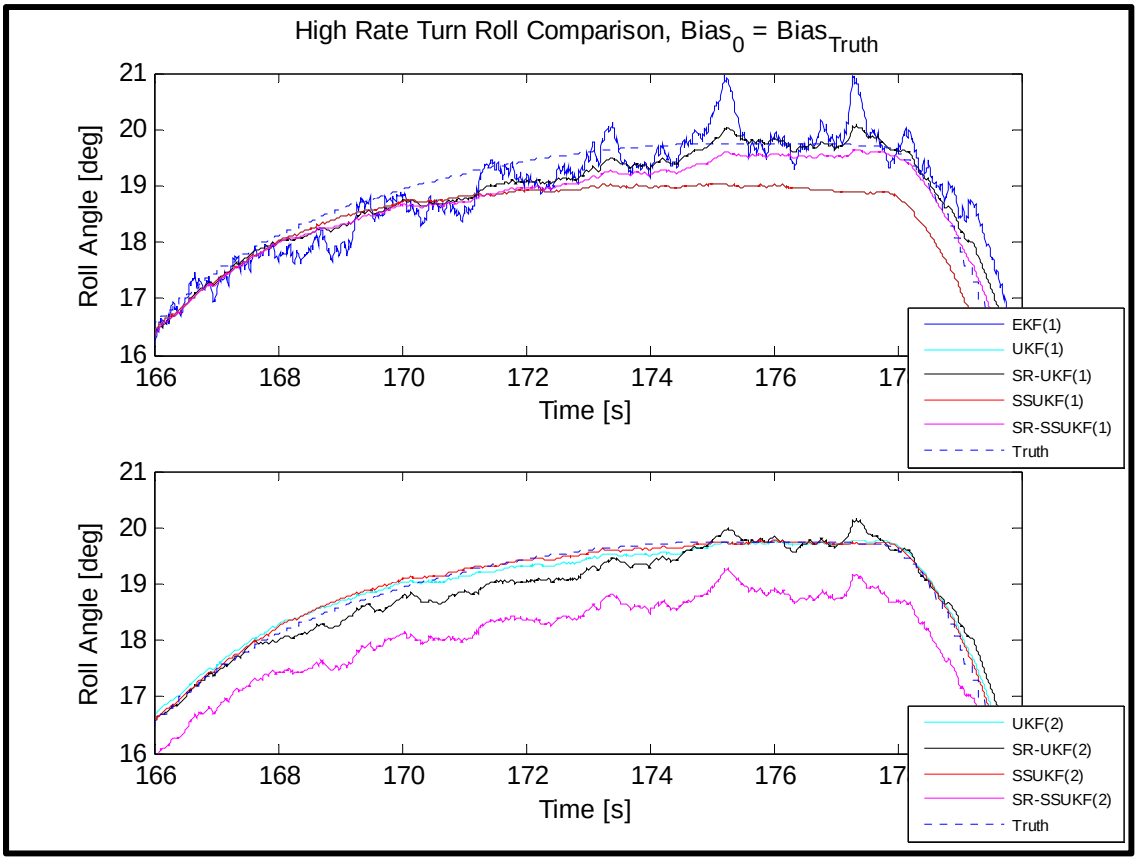

Figure 27. Roll angle estimates during a high rate turn with MM1 (top), MM2 (bottom).

The EKF(1) is the noisier of the MM1 AHRS estimators, but accurately captures the true roll value within the noise. The UKF(1) does not distinctly appear on either plot because it is practically identical to the SSUKF(1) values which is a surprising result in that the SSUKF(1) utilizes nearly half the sigma points as the UKF(1) estimator. A cleaner distinction between normal sigma points and spherical simplex sigma points is

shown in the SR-UKF(1) and SR-SSUKF(1) where the SR-UKF(1) is able to more closely follow the true roll value in high rate turns, but the SR-SSUKF(1) is slightly smoother. On the bottom plot of Figure 27 UKF(2) and SSUKF(2) perform the best in the turn nearly matching the true roll value, but have the slowest convergence time at initialization as shown in Figure 28. The SR-SSUKF(2) while containing less variance than the EKF(1) is the worst performer in the turn at nearly a degree off for the duration segment shown.

Figure 28 shows the convergence time of each estimator, again the top plot represents MM1 estimators and the bottom plot represents measurement MM2.

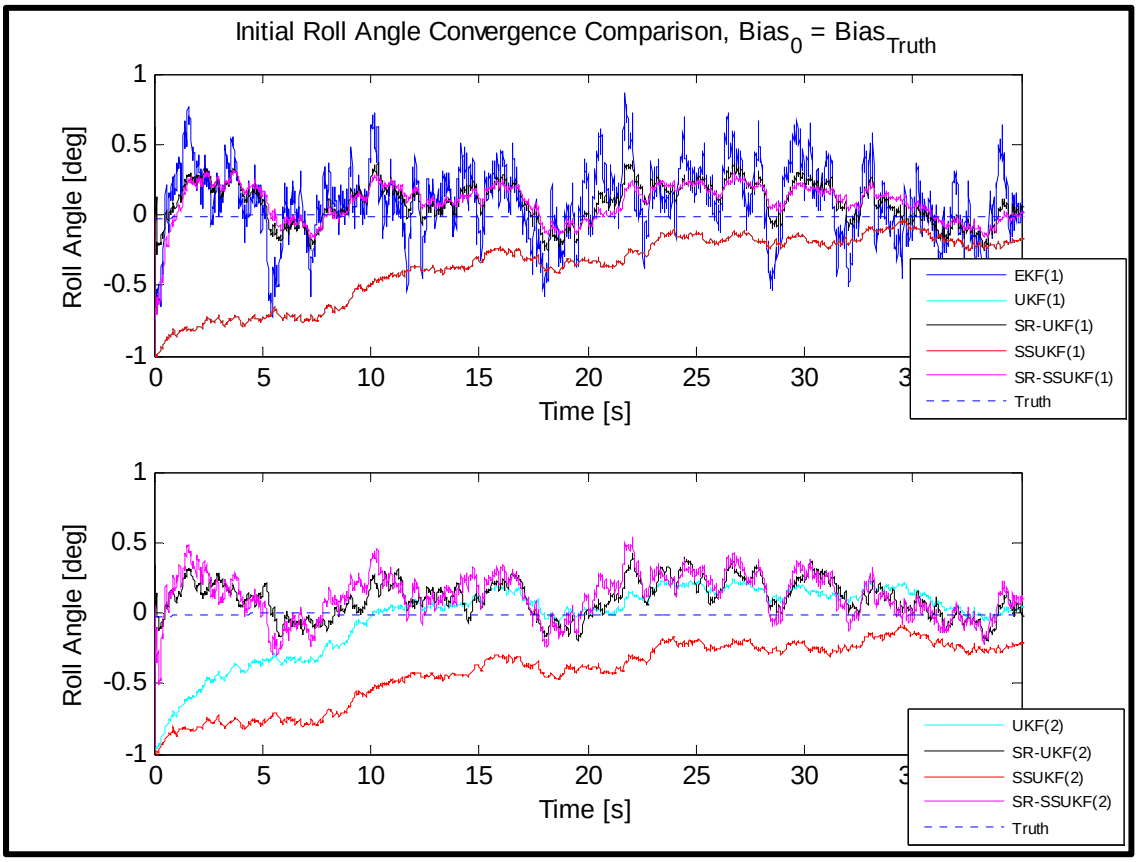

Figure 28. Roll angle estimates with MM1 (top), MM2 (bottom).

The EKF(1), SR-UKF(1), and SR-SSUKF(1) converge on the true value immediately whereas the UKF(1) and SSUKF(1) converges to within a standard deviation of the roll angle as calculated from the accelerometer approximation in AHRS MM1, after 35 seconds. Similarly the UKF(2) and SSUKF(2) have the slowest convergence time of the second measurement model group, but the UKF(2) is able to converge in 10 seconds where the SSUKF(2) converges again at 35 seconds.

Next, the pitch estimates from the first experiment are shown in Figure 29 and 30. Figure 29 contains the estimates from MM1 and two in a high rate turn, and Figure 30 shows the pitch estimate convergence.

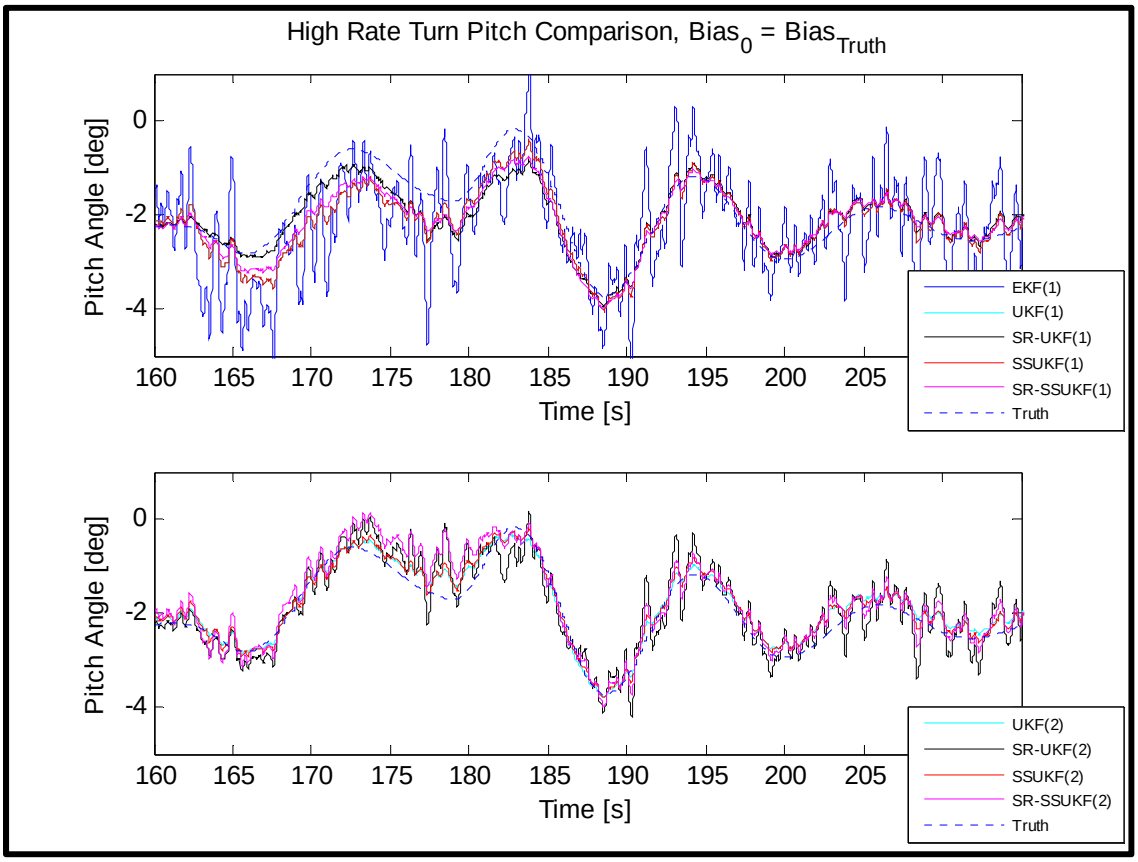

Figure 29. Pitch angle estimates during a high rate turn with MM1 (top), MM2 (bottom).

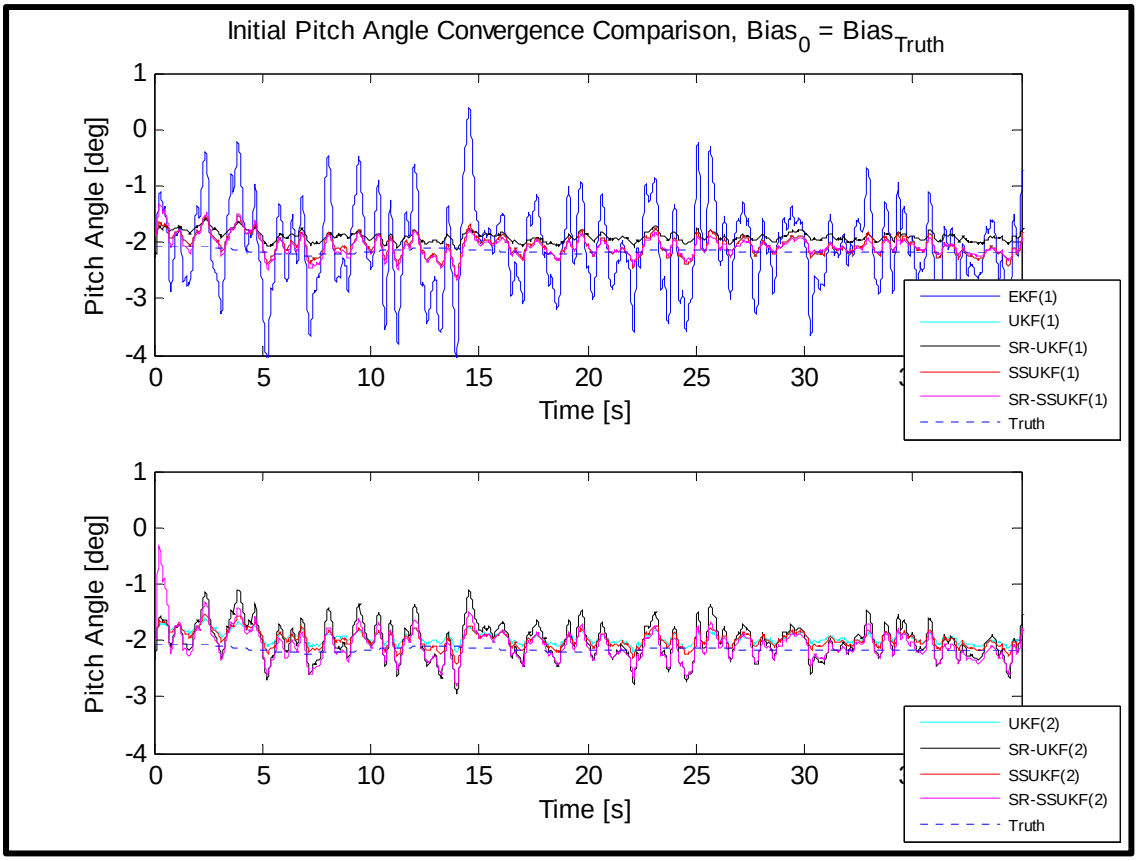

Figure 30. Pitch angle estimates with MM1 (top), MM2 (bottom).

The EKF(1) for pitch angle estimates has significantly higher variation than the unscented filters. Both sets of estimators appear to give very similar results for pitch estimation. The square root filters contain slightly more variation than the UKF(2) and SSUKF(2) filters. The biggest difference between the two measurement models for pitch estimation happens from time 170 to 185 seconds in Figure 29 where the model one estimators underestimate the pitch and the model two estimators overestimate the true pitch values. This is due to the assumptions made calculating the measured pitch values from MM1 where the Coriolis term only aids the roll measurement resulting in slightly underestimated pitch moments.

A more realistic test of each estimator's performance is with unknown gyro and accelerometer terms as in most cases the true accelerometer and gyro biases and error are rarely known without expensive bench testing. The gyro and accelerometer biases are integral to the measurement and process equations which can lead to filter divergence if the bias estimates are poor and the errors not constantly driven towards zero. Figures 31 and 32 capture the same roll angle highlights from Figures 27 and 28 to show a comparison of robustness.

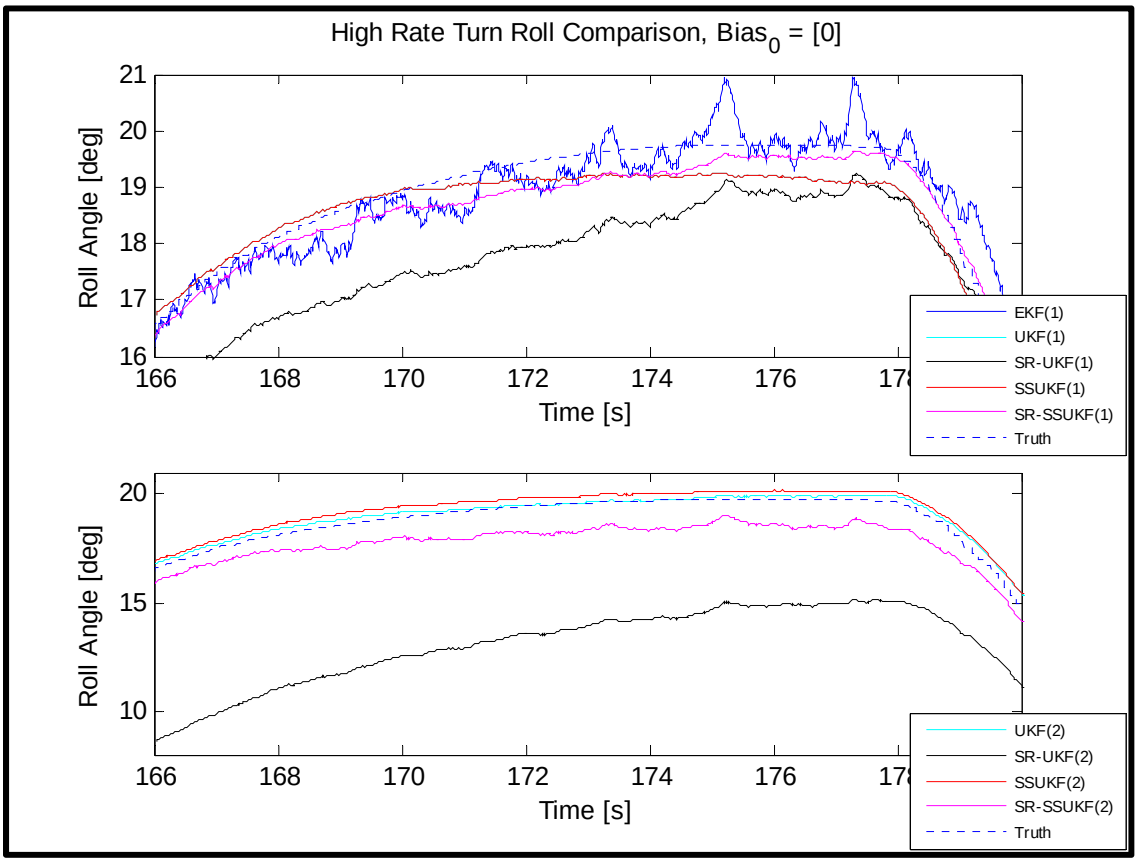

Figure 31. Roll angle estimates for MM1 estimators when gyro and accelerometer biases are unknown during a high rate turn (top), MM2 (bottom).

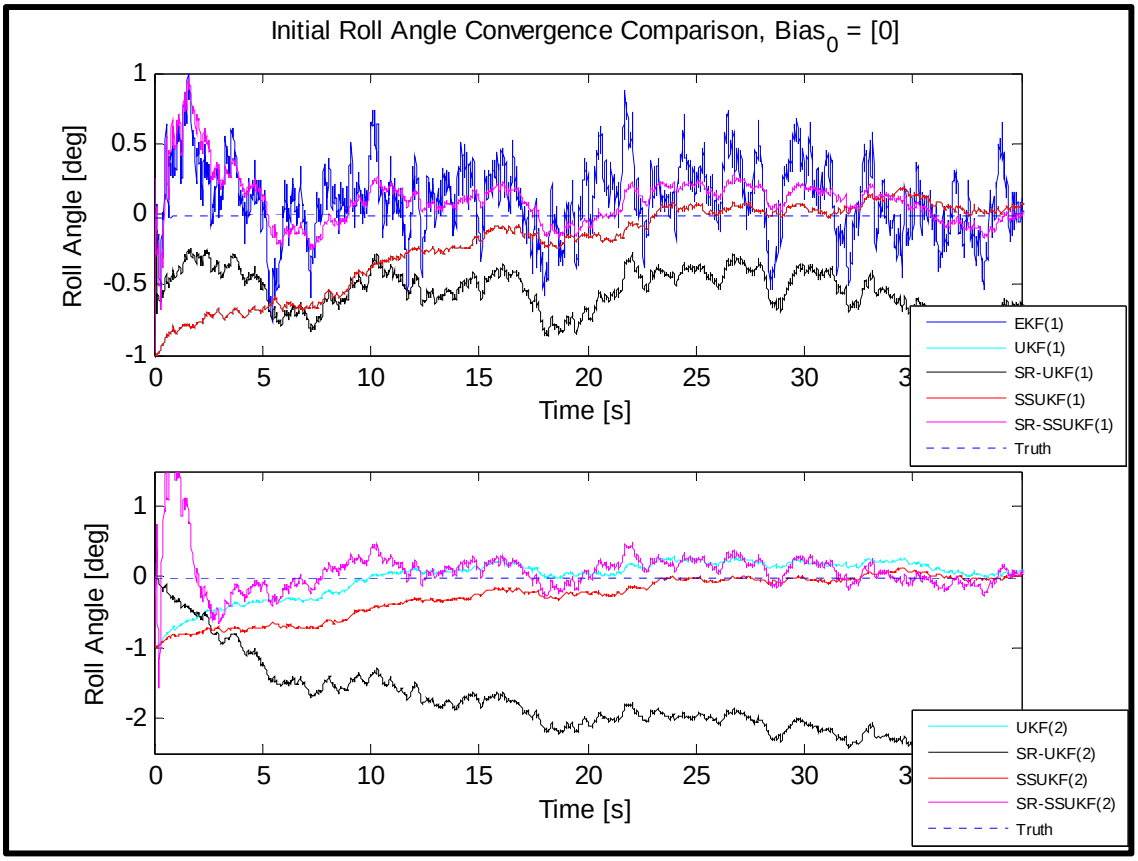

Figure 32. Roll angle estimates for MM1 estimators when gyro and accelerometer biases are unknown (top), MM2 (bottom).

Each filter displayed slightly less accuracy than in the known bias case, but surprisingly the SR-UKF which for the known bias case performed the best, for both measurement models performed the worst and slowly diverged in the SR-UKF(2) implementation.

The pitch estimates, shown in Figures 33 and 34, follow a similar trend to the roll estimates in that for the most part all the estimators are slightly less accurate, but the SR-UKF for both measurement models performs the worst.

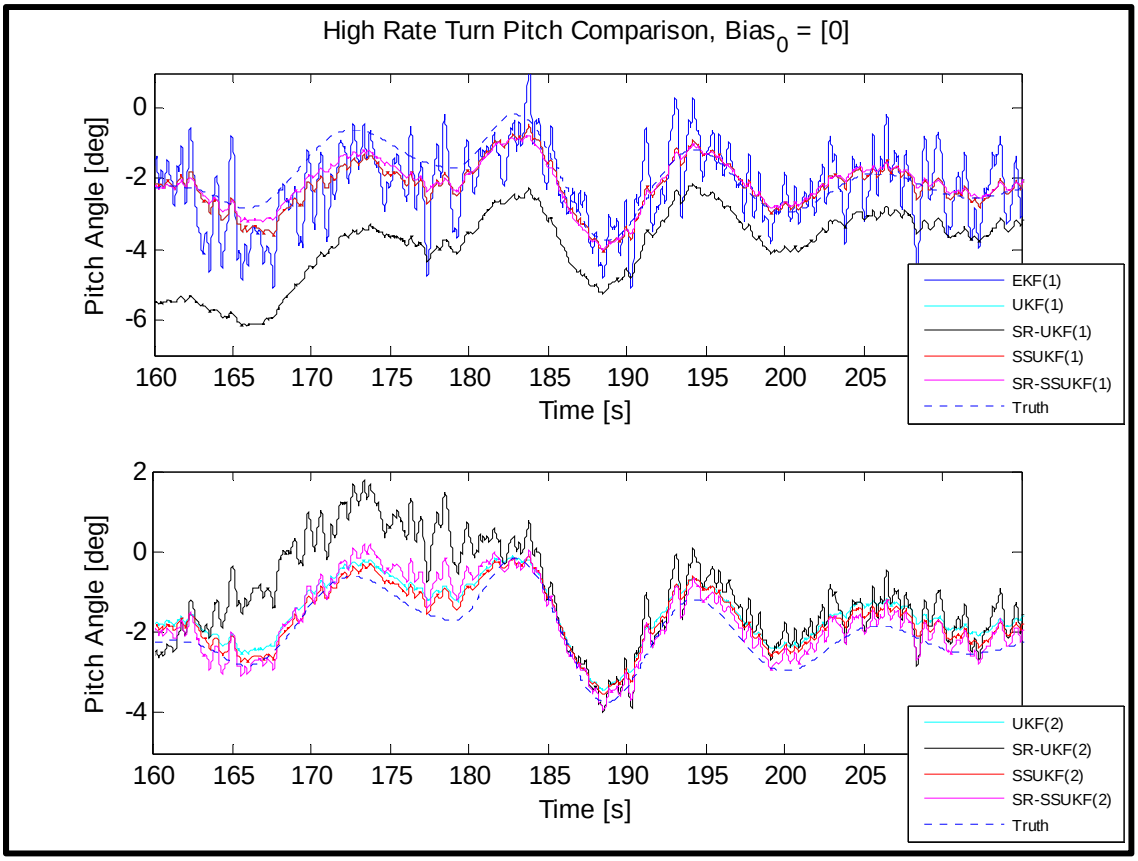

Figure 33. Pitch angle estimates for MM1 estimators when gyro and accelerometer biases are unknown during a high rate turn (top), MM2 (bottom).

In Figure 33, the SR-UKF(1) pitch estimates initially diverge, then slowly begin to converge during the high rate turn which indicates that the bias' are not diverging for the entire duration of the data set.

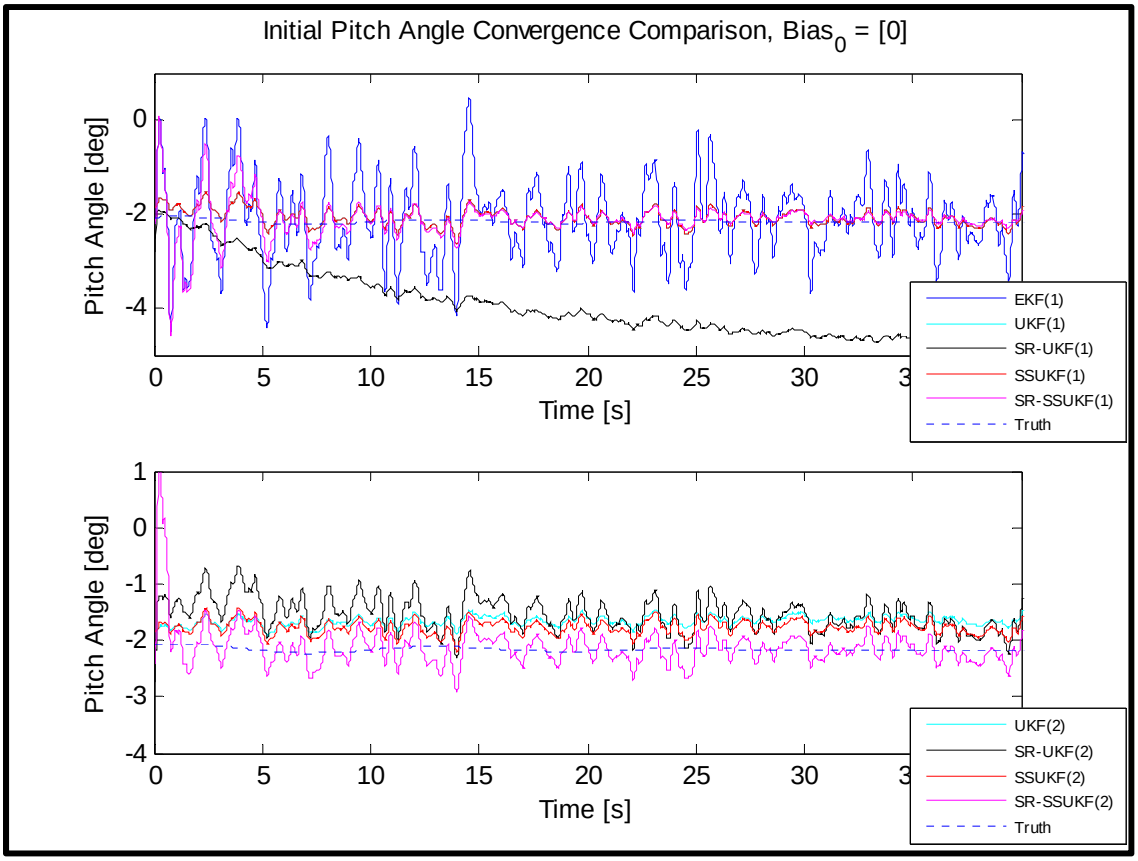

Figure 34. Pitch angle estimates for measurement model 2 estimators when gyro and accelerometer biases are unknown (top), MM2 (bottom).

The relatively poor performance of the SR-UKF estimator and the difference between the two variants is explained best by viewing the estimated bias error in Figure 35. The AHRS module for both measurement models largely depends on accurate gyro bias estimation. Figure 35 shows the root mean squared (RMS) error of the residual between the true gyro bias and estimated bias.

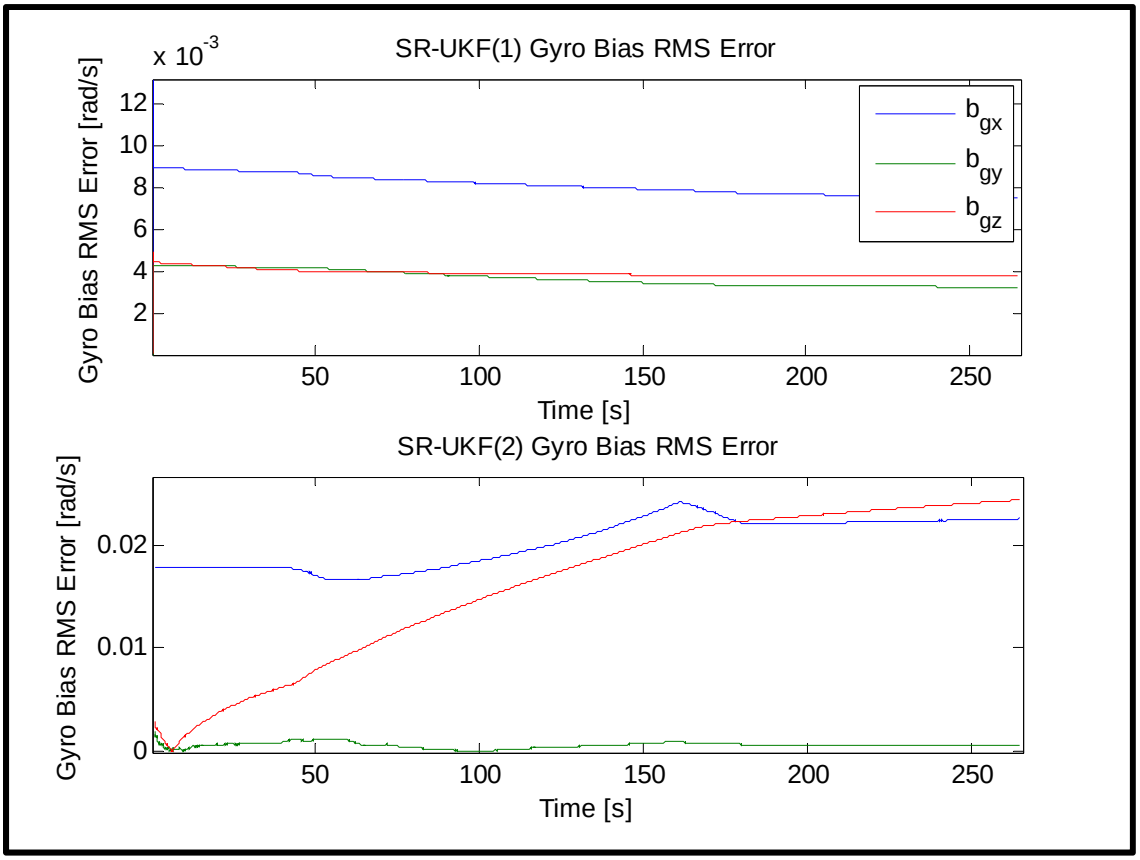

Figure 35. SR-UKF gyro bias estimates for experiment 2.

The gyro bias error for SR-UKF(1) is slowly decreasing throughout the data set, but remains an order of magnitude off which explains the constant offset seen in the pitch and roll estimates. The bias errors for SR-UKF(2) are two orders of magnitude off in the roll and yaw rate channels and growing which explains why the roll estimate constantly diverged. The pitch rate channel bias error of SR-UKF(2) remains at a low value on the same order of magnitude as the actual bias which explains why it was able to track pitch nearly as well as the other estimators.

The average RMS error for pitch, roll, and yaw for both experiments is summarized in Figure 36. The EKF(1), UKF(1), SSUKF(1), and SR-SSUKF(1) estimators were the most robust, gaining the least amount of error from arbitrary bias initialization.

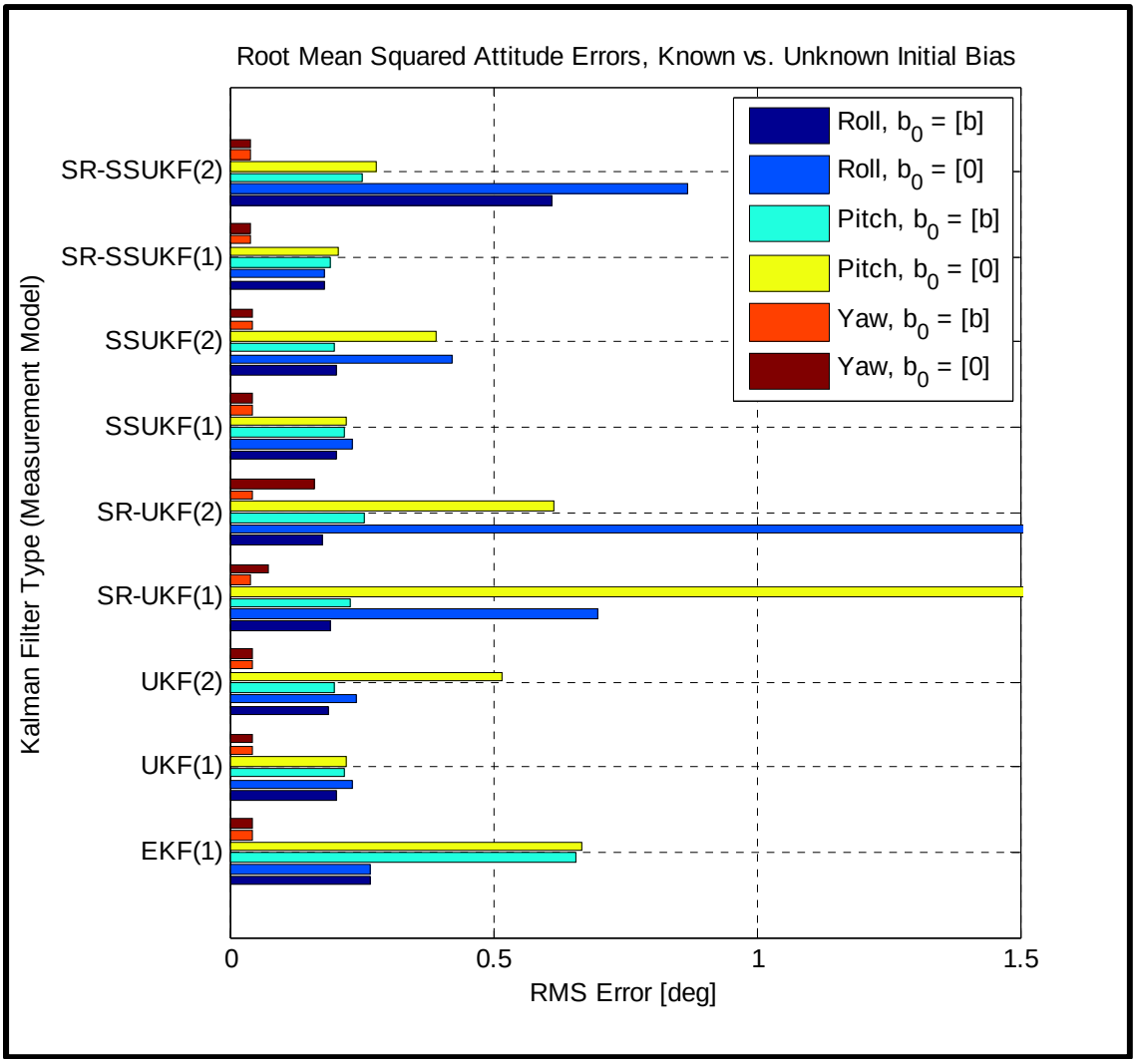

Figure 36. Summary of averaged RMS error for attitude estimation for both experiments.

The EKF(1), while robust in comparison, had the highest RMS values for pitch and yaw, and second highest for roll when the bias was well known. The SR-SSUKF(1) had the best overall performance claiming the lowest RMS error for pitch, roll, and yaw when the bias was unknown, and second lowest when the bias values were known at initialization.

E. POSITION AND VELOCITY PERFORMANCE

For Condor simulation the INS process model equations were simplified by assuming a local tangent plane frame of reference and no sidereal rate, which created a more linear process. The measurement model shared by all the estimators assumes the GPS position and velocity measurements are corrupted with Gaussian white noise, but no other errors, which in the case of the Condor simulation environment is true. The RMS velocity errors from the known and unknown bias experiments are shown in Figure 37.

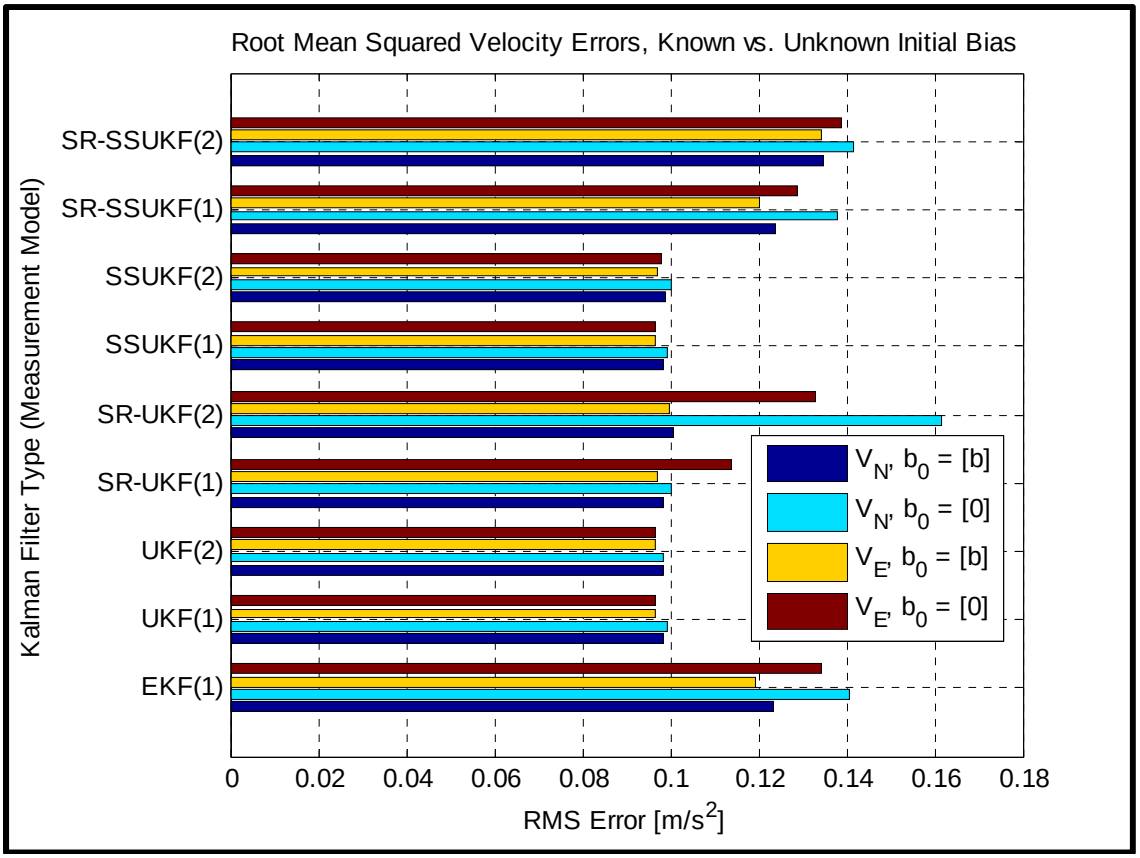

Figure 37. RMS velocity estimate errors.

Overall, the RMS values for each filter are near identical which is to be expected for the relatively clean aiding GPS measurements. The unknown bias case for SR-UKF(2) is the worst performing estimator which can be explained by both the poor attitude estimates the INS module received and the relatively large accelerometer bias errors shown in Figure 38.

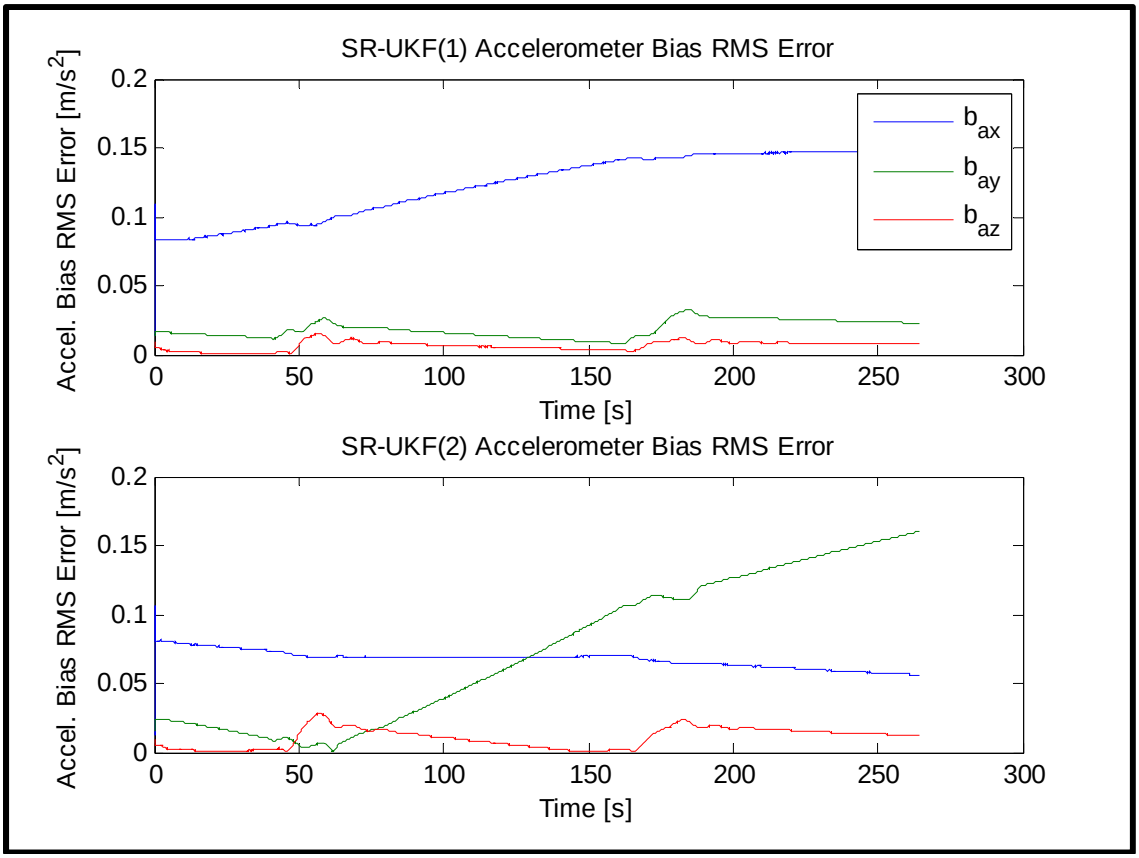

Figure 38. Accelerometer RMS bias errors for SR-UKF(2) with arbitrary initial bias values.

Despite the relatively similar accuracy in velocity estimation there is a larger difference in position estimation shown in Figure 39. The position estimates are purely a function of the current velocity and aiding sensor measurement. The aircraft in Condor simulation soars at near 30 m/s, so small errors in velocity can quickly cause the position estimate errors to grow.

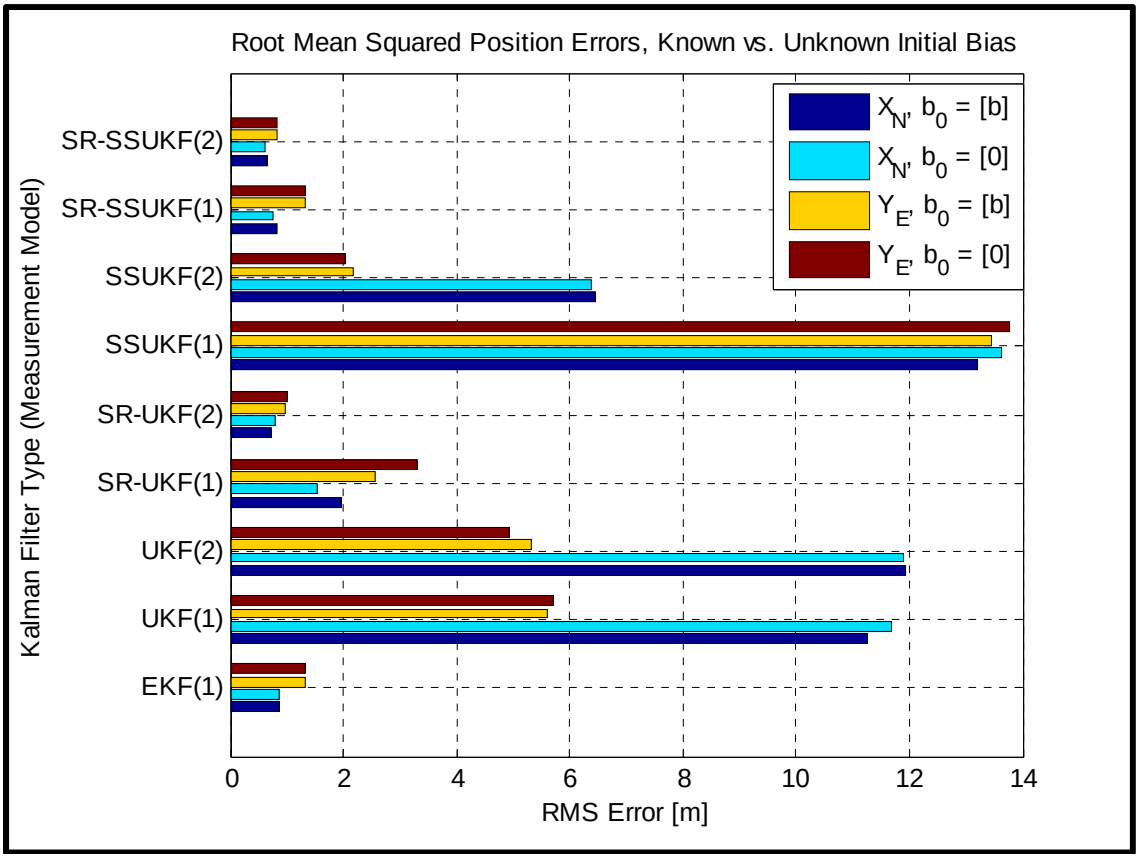

Figure 39. RMS position estimate errors.

None of the position estimate errors grew out of control which was largely a function of proper filter tuning. The lack of accurate bias and attitude estimation in the case of the SR-UKF had little effect on position quality as the estimator was able to overcome the error by relying more heavily on the GPS measurement.

F. ESTIMATOR RANKING AND SELECTION

To determine the best estimation filter for this navigation system design, each estimator was ranked based on a one-to-nine scale base on the average RMS error (Figures 36, 37, and 39) and the relative computation times (Figures 25 and 26). Figure 40 is a radar plot of the estimator rankings in each major category: accuracy, robustness, and efficiency.

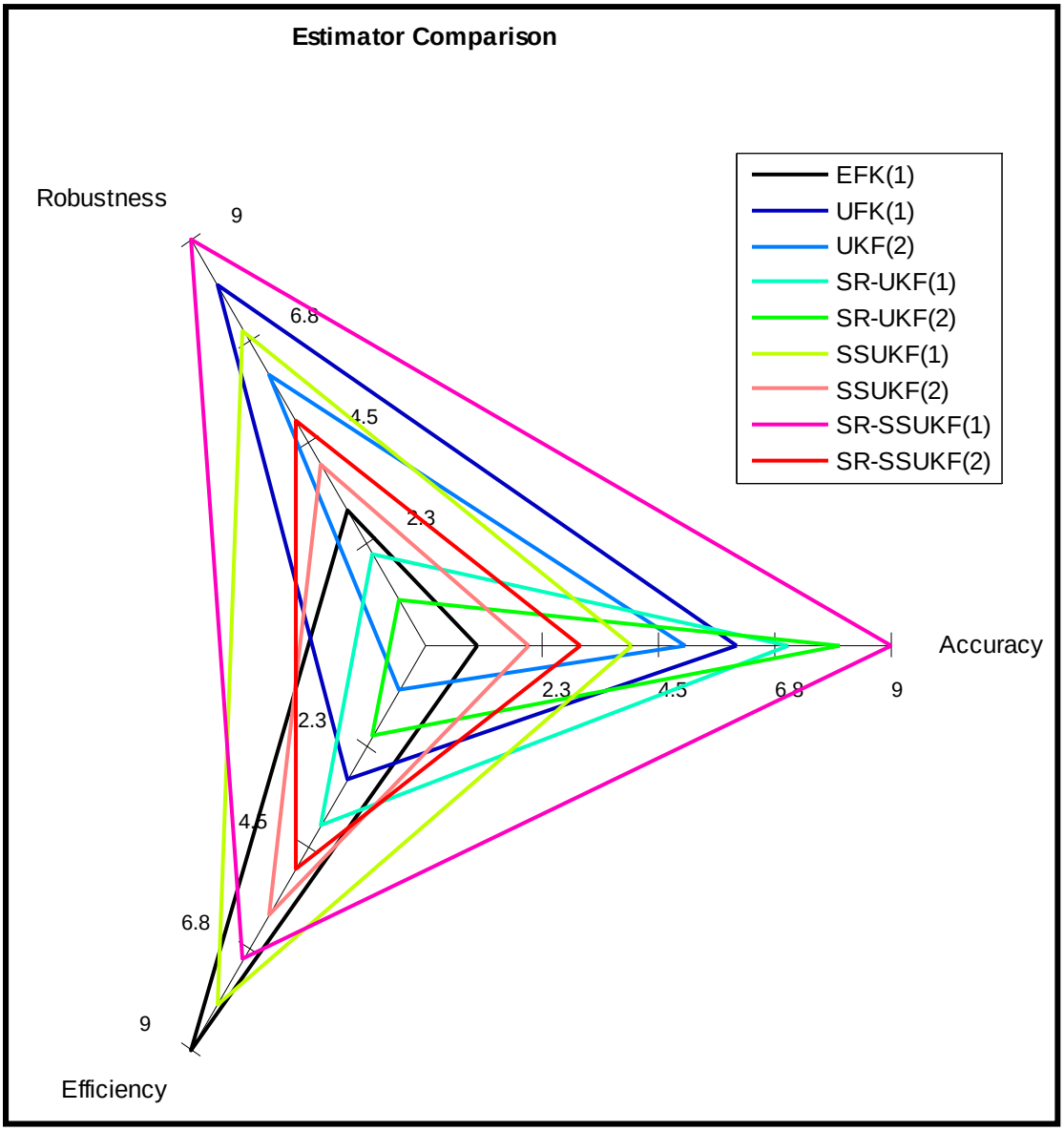

Figure 40. Radar plot of filter evaluation criteria, score on a scale of one to nine, where nine is the best score.

The first experiment, where the gyro and accelerometer bias was initialized with the true values, represents an evaluation of accuracy and takes into account attitude, position and velocity RMS values. An evaluation of robustness is determined based on the average RMS values from the second experiment where bias values where initialized to zero and also takes into account attitude, position, and velocity errors. The efficiency

category takes into account the relative speed of each UKF in comparison to the EKF runtime.

In general, the SR-UKF variants were the most accurate, followed by the standard UKF and SSUKF variants, respectively, while the EKF ranked last. Moreover, filters based on MM1 were typically more accurate than those based on MM2. Surprisingly, the SR-SSUKF did not follow the same trend. This filter appeared to be especially sensitive to the choice of measurement model, as SR-SSUKF(1) was the most accurate filter of all, while SR-SSUKF(2) was the least accurate UKF variant. The EKF last place ranking was expected since it relies on the linearization of the highly nonlinear measurement equations of the AHRS module.

A compelling result from these experiments is that the SR-SSUKF(1) was both the most accurate and most robust filter tested. In general the estimators utilizing AHRS MM1 out performed the MM2 variants. SR-UKF(2) was the least robust as the errors for attitude diverged for the duration of the data set in experiment two. The EKF, while near the bottom of the pack fared well in that the RMS errors saw little change with arbitrary bias initialization, but the relative estimation accuracy was limited by the high variation in each estimate.

With respect to efficiency, the EKF surpasses all the UKF variants. As previously discussed, the UKF is dominated by the number of iterations per time step leading to the high ranking of the spherical simplex variants and the low ranking of the normal weighted UKF.

Direct comparison is an important consideration when evaluating the qualities of each filter variant. For the SEAFOX II application however, it is more important that the candidate estimator perform well in specific categories. Table 10 lists the subjective design weights which were used as multipliers against the relative ranking of each filter.

Ranking Weights		
Parameter	**Accuracy Experiment Weights**	**Robustness Experiment Weights**
Roll	9	10
Pitch	9	10
Yaw	3	4
X_N	5	6
Y_E	5	6
V_N	4	5
V_E	4	5
Time	10	10

Table 10. Subjective weights assigned to each evaluated estimation category.

The weights scale from 1–10, where ten is the most critical and 1 is the least critical. The COMNAV GPS system onboard SEAFOX II provides accurate position, speed, and heading estimates at a frequency of 20 Hz, which exceeds the needs of the ATLAS Sonar. What is not provided is accurate pitch and roll estimates, which are critical parameters for sonar correlation accuracy. Lastly, the PC-104 installed on SEAFOX II will be running multiple programs for various systems; therefore the efficiency of the estimator is very important. The results for the weighted comparison are shown in Figure 41.

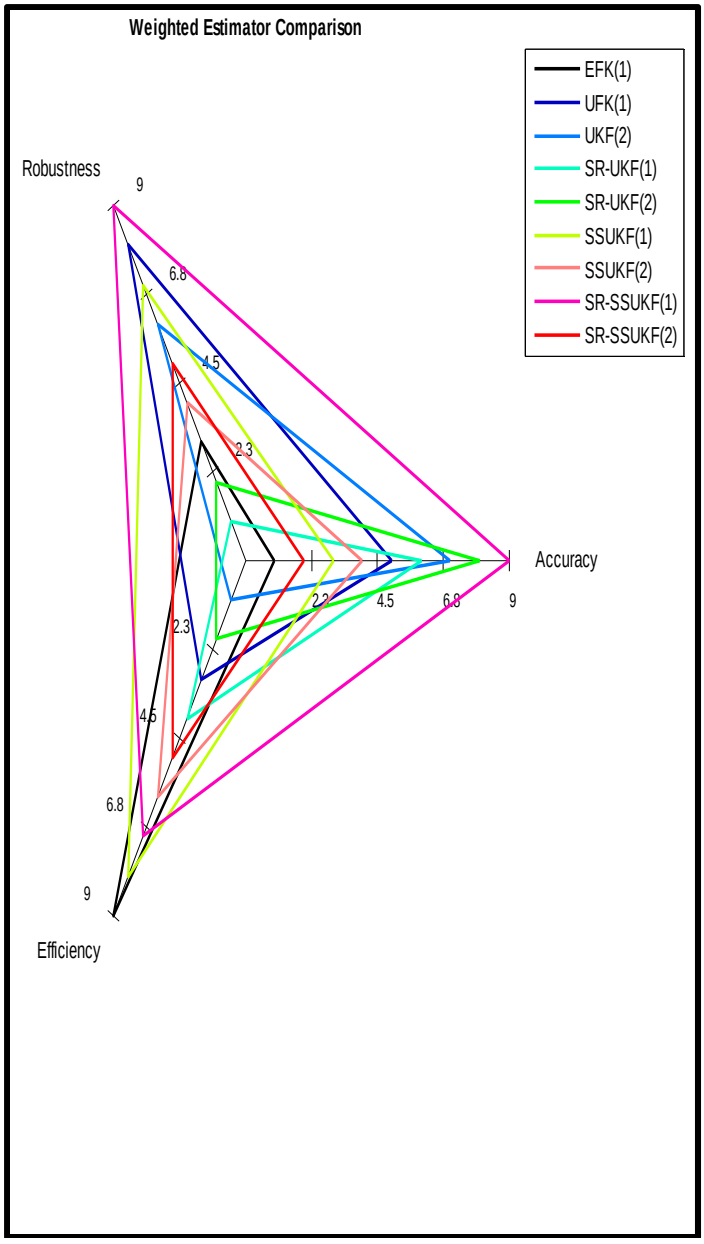

Figure 41. Weighted estimator comparison.

The results for accuracy are slightly shuffled to highlight the filters with the best attitude estimation. These results compare directly with the RMS error plot previous shown in Figure 36 where the MM2 variants of each estimator slightly outperform their model one counterparts with the exception of the SR-SSUKF(1) which outperforms them all. The results for robustness saw no change as the most robust filters provided the best attitude estimates. The efficiency category also saw not change as the category was

simply scaled by the weighting factor. Lastly, the weighting ranking of each category was summed for each filter to provide an overall score where the lowest score is the best performing filter within the weighting scheme from Table 10. The results for total score are shown in Figure 42.

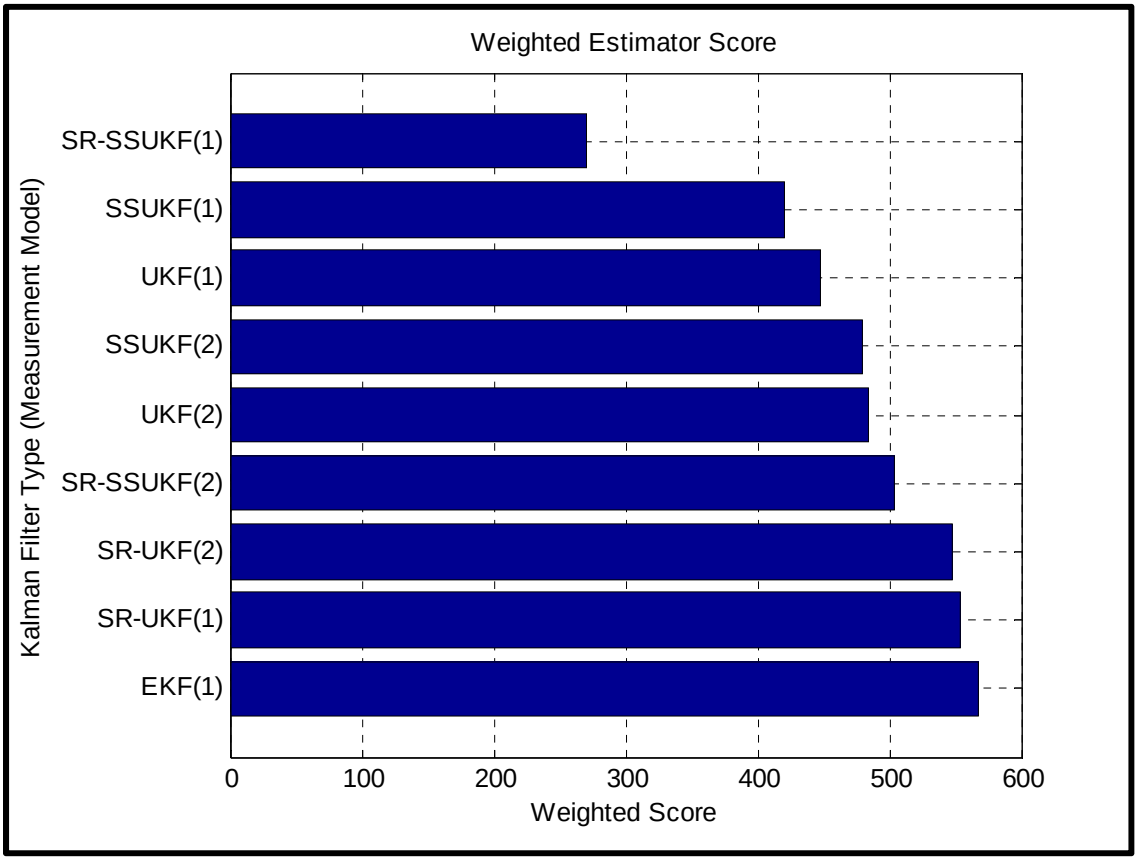

Figure 42. Weighted estimator score based on RMS errors and design weights. Lowest score is best performing filter.

Despite the top spot for computational efficiency, the EKF was near the bottom of the pack for the accuracy and robustness category. In this navigation estimator design, the EKF performed well for velocity and position estimation, but those performance categories were discounted in comparison to the need for robust and accurate attitude estimation. With the best weighted and un-weighted accuracy and robustness as well as efficient computational complexity, the SR-SSUKF(1) is clearly the best choice for implementation onboard the SEAFOX II.

G. SEAFOX II ESTIMATION COMPARISON

For this last experiment, data was collected from sea trial maneuvers on the Monterey Bay. The portion of data used to compare the EKF(1) and SR-SSUKF(1) was captured during zig-zag and turning circle maneuvers which provided relatively large acceleration in the turns. Figure 43 compares position estimates and highlights the portion of the data where the SEAFOX II was executing constant rudder turning circles.

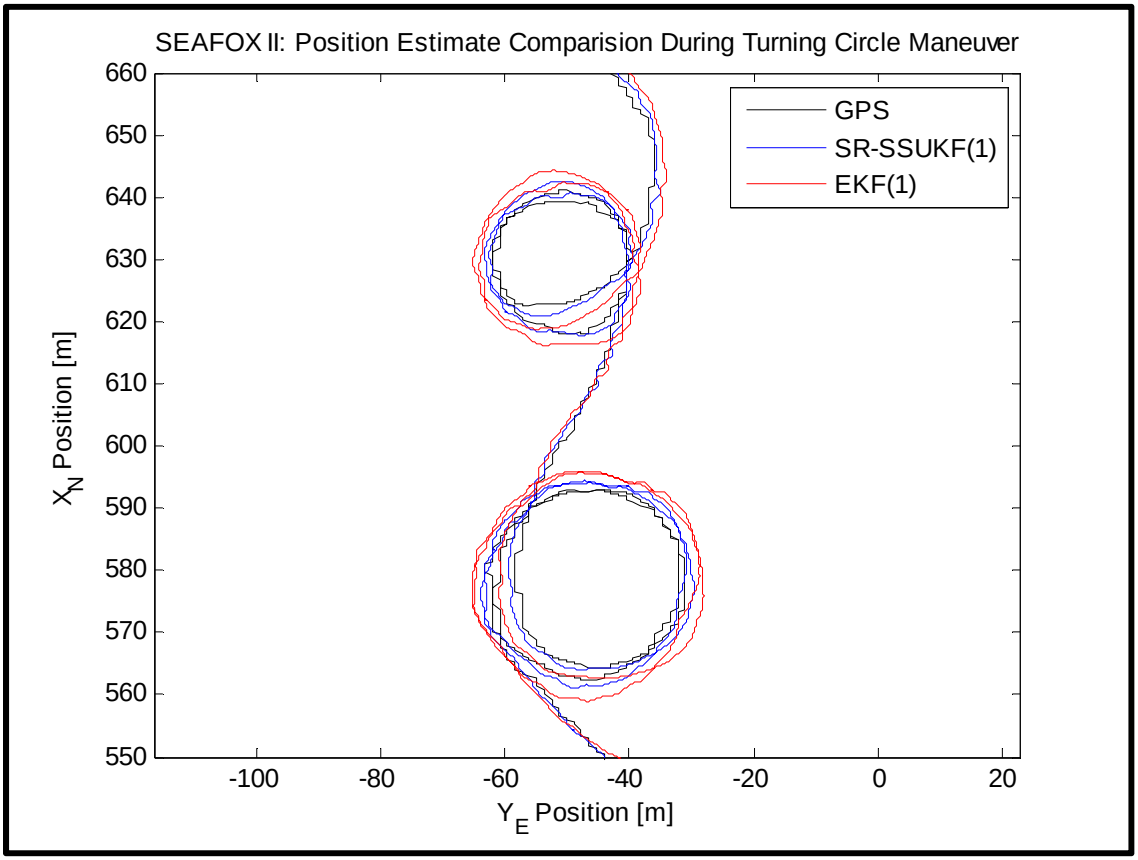

Figure 43. EKF(1) and SR-SSUKF(1) position estimate comparison during turning circle maneuvers.

On the day on which this particular data set was obtained, the primary GPS receiver was down for maintenance and the secondary receiver provided the position, speed, and course over ground. This particular receiver updates at 5 Hz, compared to the normal 20 Hz output by the primary receiver. Both EKF(1) and SR-SSUKF(1) were

able to maintain accurate position estimates. The SR-SSUKF(1) was able to track slightly closer to the GPS measurement. North and east velocity components are shown in Figure 44.

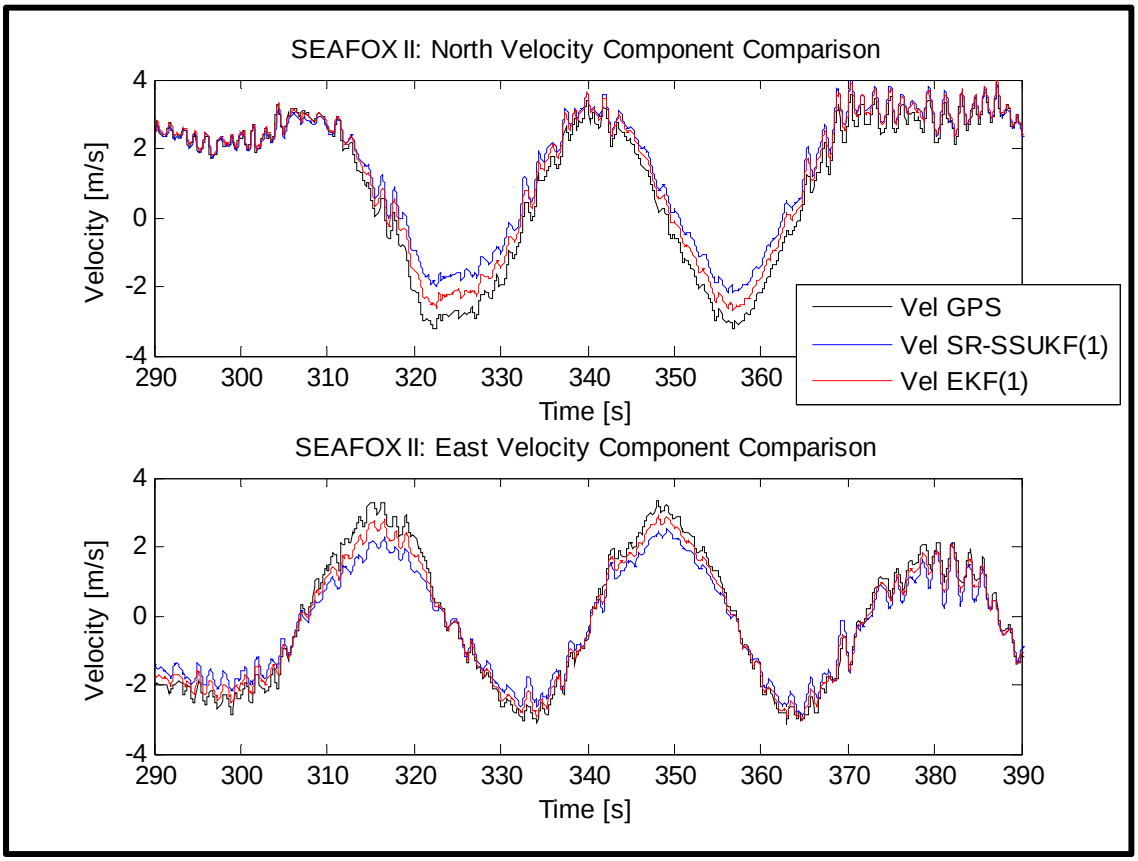

Figure 44. EKF(1) and SR-SSUKF(1) velocity estimate comparison during turning circle maneuvers.

The EKF(1) in Figure 44 tracks closer to the GPS velocity measurement than the SR-SSUKF(1), but this is mainly due to the tuning of the SR-SSUKF(1) measurement noise matrix. By forcing the SR-SSUKF(1) to more closely mimic the GPS velocity measurement, the position estimate degraded. Slightly increasing the velocity channel measurement noise gains actually increased the position estimation accuracy of the SR-SSUFK(1) without introducing excess variation.

Attitude estimation for SEAFOX II is the most critical component of this particular navigation system design as its sensors do not directly measure roll angles. The comparison plots in Figures 45 and 46 show the pitch and roll estimates of the SEAFOX II during the turning circle maneuver. Both estimators are compared against the unfiltered gyro only solution of pitch and roll. For the short duration under consideration, the SEAFOX II military grade ring laser gyro provides a highly accuracy approximation of pitch and roll values.

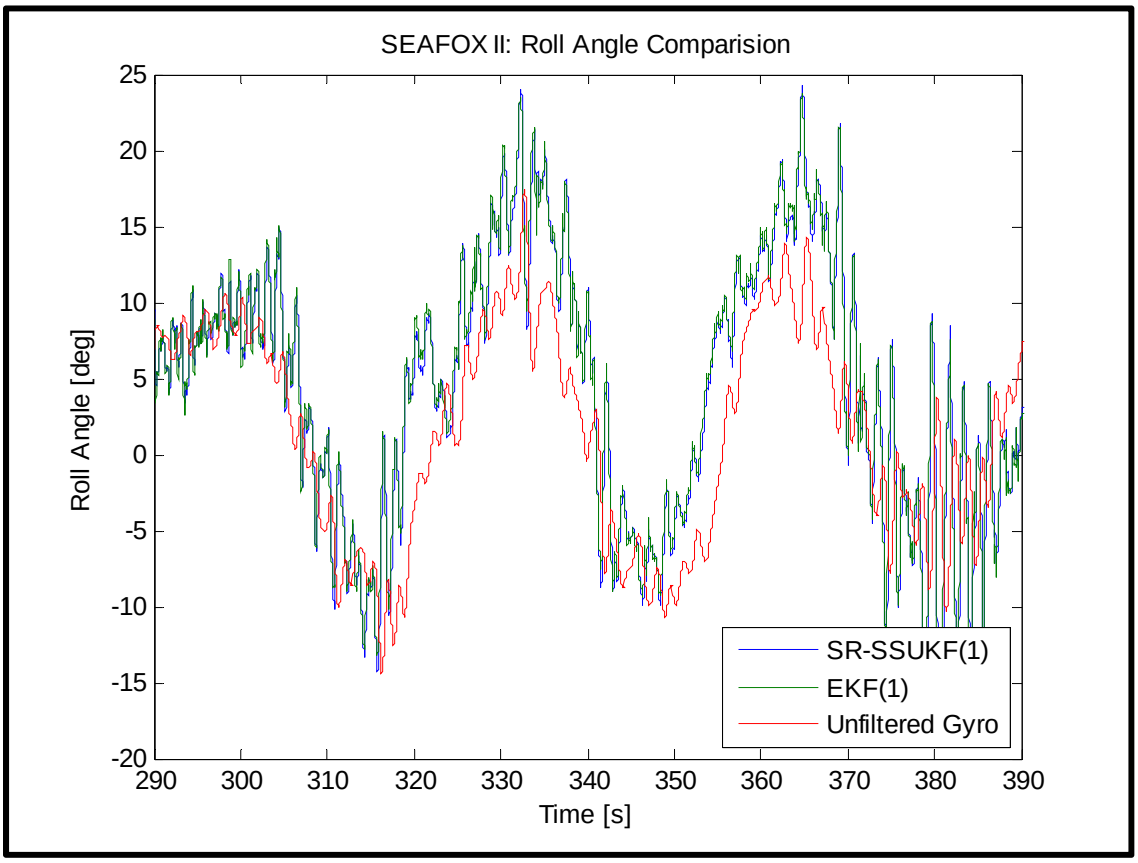

Figure 45. EKF(1) and SR-SSUKF(1) roll estimates compared against unfiltered gyro integration.

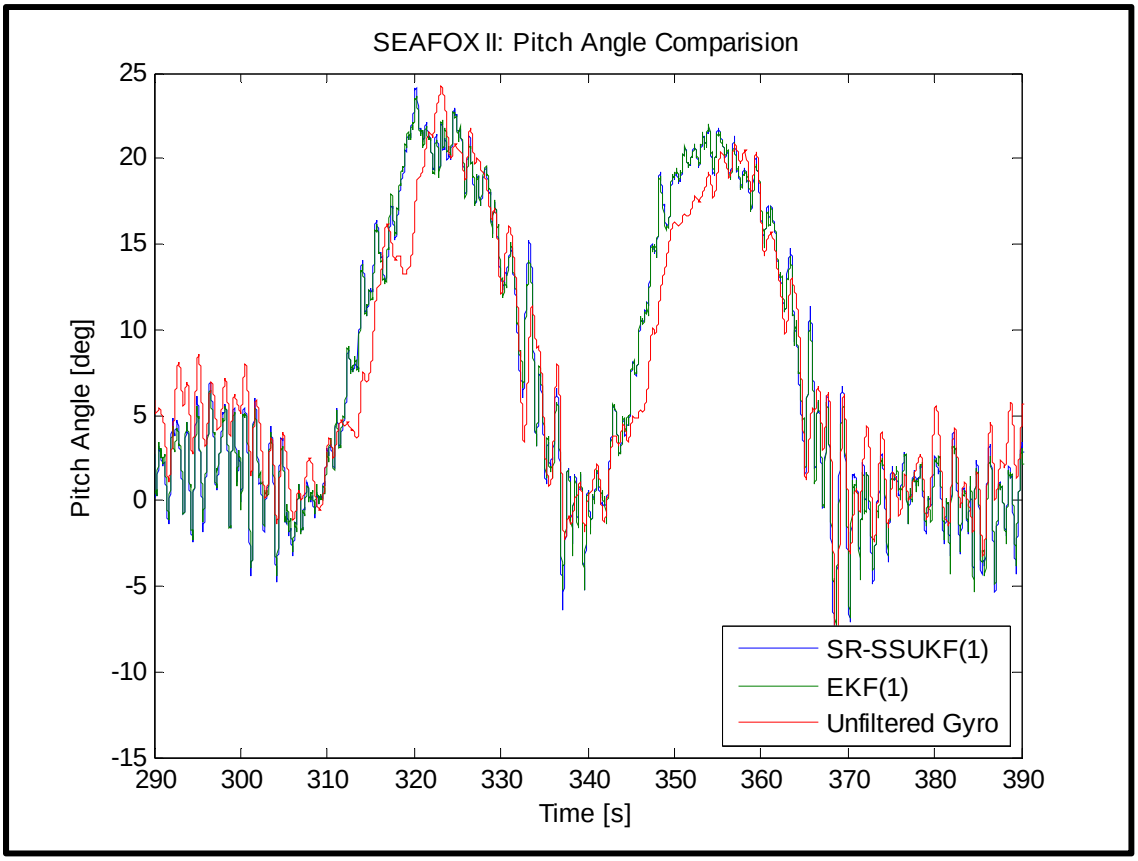

Figure 46. EKF(1) and SR-SSUFK(1) pitch estimates compared against an unfiltered gyro solution.

The large values of roll and pitch may at first seem excessive for a boat operating near shore, but there were significant surface wave and swell action on the day this particular data set was gathered. The EKF(1) and SR-SSUKF(1) produced near identical results for pitch and roll estimation. Both filters appear to slightly overestimate the roll values near the peaks in Figure 45, but provide good estimates of pitch in comparison the to the unfiltered gyro solution.

From these plots it is apparent that there is little difference in the overall accuracy between the EKF(1) and SR-SSUKF(1) when considering low speed turning maneuvers on the water. The EKF in general is a tried and true filter that was expected to perform well for this navigation estimation task. By using the local tangent plane frame of reference, the navigation equations were simplified resulting in a more linear process model. The two measurement models tested were both nonlinear, but the second model

which estimated the accelerometer measurements was too nonlinear for the EKF, resulting in filter divergence. The unscented Kalman filters, on the other hand, are designed to address the problem of nonlinear estimation by propagating an iterative set of points through the nonlinear process to approximate the true probability distribution and recover its mean and covariance. This process, while accurate, comes at a high computational cost and may be too high for some systems. In an effort to reduce the computational burden associated with the UKF process, the navigation estimator was split into two modules, reducing the number of states per estimator and therefore reducing the number of iterations required by the UKF. For the navigation estimator designed here, the benefits of the EKF are accuracy and robustness on par with the unscented Kalman filters, but with less computational burden. Detracting from the EKF is the requirement to compute large Jacobian matrices of partial derivatives, a potentially large source of error during software implementation as the manual coding of these equations is extensive. The SR-SSUKF operates at a low computational cost in comparison to other UKF variants, but still requires nearly double the computation time of the EKF. Perhaps more importantly, the SR-SSUKF provides equal or better accuracy than the EKF with more robustness, but does not require coding or computing a single derivative matrix, making it easier to implement in software than the EKF.

THIS PAGE INTENTIONALLY LEFT BLANK

VIII. CONCLUSION AND RECOMMENDATIONS

A. CONCLUSION

The SEAFOX II's proprietary architecture initially limited its ability to perform experimental, fully autonomous operation, thereby relegating its utilization a chase and recover boat for other autonomous systems. In order to build autonomous capabilities into the SEAFOX II and better support the installation of the bow mounted ATLAS sonar, the proprietary remote control systems onboard were replaced with an open architecture network of sensors, controllers, and computers. While the SEAFOX II was capable of producing the standard maritime navigation aids of heading, speed, and position, it did not have an ability to produce accurate pitch and roll estimates the ATLAS sonar requires to produce accurate sonar maps. To meet this need, the extended Kalman filter and four variants of the unscented Kalman filter were tested in a high fidelity simulation environment to determine the navigation estimator with the best combination of accuracy, robustness, and computational efficiency. In an effort to reduce computational complexity given that the SEAFOX II has limited computing resources, the navigation estimator was separated into two modules, one estimating vessel attitude (AHRS) and the other estimating position and velocity (INS). Two complex nonlinear measurement models were investigated to compliment the attitude estimator. The first measurement model provided instantaneous estimates of pitch and roll by combining measurements from the accelerometer, gyro, and acceleration estimates based on measured GPS velocity. The second measurement model recreated accelerometer measurements by fusing the AHRS and INS state estimates with estimated GPS accelerations. Both models provided accurate attitude estimation, but the second model proved too nonlinear for the extended Kalman filter to converge. After extensive simulation and testing the square root spherical simplex unscented Kalman filter (SR-SSUKF) combined with the AHRS measurement model that approximated pitch and roll measurements exhibited the best combination of accuracy, robustness, and efficiency.

The SR-SSUKF estimator was then compared against the EKF for evaluation of SEAFOX II data. While the SR-SSUKF significantly outperformed the EKF in the simulation environment, the two estimators provided near identical estimation results when evaluating actual sea trial data.

In conclusion, for the problem of navigation state estimation on a slow moving small boat operating in local waters, the extended Kalman filter provides adequate attitude, position, and velocity estimates with relatively low computational overhead. In simulation with highly dynamic maneuvers, the EKF consistently underperformed in comparison to nearly all the unscented Kalman Filters for accuracy and robustness. The SR-SSUKF on the other hand proved to be a powerful tool that provided equal to or better accuracy than the EKF in both low and high dynamic maneuvers at a relatively low computational cost with the added benefit of software implementation simplicity. The SR-SSUKF is highly recommended for use onboard the SEAFOX II.

B. RECOMMENDATIONS FOR FUTURE WORK

The Kalman filter estimators provided good navigation state estimates over short time periods, but were not tested over long periods of time. It is recommended that each estimator be tested over longer durations to insure that the bias estimates do not diverge, causing the entire states estimate to diverge. While both the EKF and SR-SSUKF tested actual SEAFOX II data, neither estimator had been implemented onboard SEAFOX II's PC-104 hardware for real-time at-sea evaluation.

The ATLAS sonar will provide a unique opportunity to combine path generation and path following algorithms with live sonar data for submerged obstacle avoidance, target detection, and target following. The REMUS 100 AUV could be used as a target for the SEAFOX II to track and follow. This would be a difficult navigation task as the ATLAS sonar has a limited field of view and range, meaning the SEAFOX II would need to navigate efficiently to keep the REMUS within the limits of the ATLAS sonar's detection region.

The UKF proved itself an accurate state estimator, but it can also be modified for use in parameter estimation. The SEAFOX II is effectively a black box in terms of steering and speed model system identification. Producing an accurate model of the SEAFOX II's steering and speed dynamics would prove invaluable in the creation of advance rudder and throttle controllers. Along the same lines, adaptive controllers could be developed that forego the need for any system identification modeling. This would be especially useful as the SEAFOX II often operates in rough coastal waters where the large wave action is difficult to remove from rudder and engine feedback controllers.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX: MATLAB CODE

A. OVERVIEW

Contained in the subsequent sections is the code used to evaluate each Kalman filter in this thesis. The MATLAB code is designed as a function and intended for use in Simulink in discrete time with fixed intevals of 100 Hz. The code was set up to enable either Condor data filtering or SEAFOX II filtering which changes certain assumptions about the LTP frame, specifically the inclusion of the sidereal rate in the SEAFOX II implementations. To toggle between data types, the last function input (u) must be toggled a 1 for Condor data or a 2 for SEAFOX II data. The code was implemented in Simulink via an *Interpolated MATLAB function* block.

B. EXTENDED KALMAN FILTER MATLAB CODE

1. Measurement Model One Implementation

```
function X_HAT = EKF1(u)
%% Extended Kalman Filter for Navigation State Estimation
% Measurement Model 1
% LT Steven Terjesen
% September 2014

% This estimator is built for use in MATLAB Simulink. There are 38
inputs
% required and can be run in Simulink with an "Interpolated MATLAB
% Function" block. The first 13 inputs are the vehicle data: Course
Over
% Ground, Speed Over Ground, Accelerometer Measurements, Gyro
Measurements,
% GPS Measurements (In LTP XNorth-YEast-ZDown), Heading Measurement,
and
% Vertical Velocity (zeroed out for SEAFOXII data). Inputs 14 through
34
% are the Process Noise and Measurement Noise diagonal elements. These
% elements are not hard coded to allow for easier tuning. Inputs 35
% through 37 are the N-E-D accelerations in the LTP frame. Input 38 is
a
% toggle for selecting whether the data is from Condor or SEAFOX II.

%% System Inputs
% Measurement inputs
CoG = u(1);                                    %GPS Course Over Ground
speed = u(2);                                  %GPS Speed Over Ground
fx = u(3); fy = u(4); fz = u(5);               %IMU Accelerations
p = u(6); q = u(7); r = u(8);                  %IMU Angular Rates
```

```
N_sat = u(9); E_sat = u(10); D_sat = u(11);     %GPS Position (NED)
heading = u(12);                                %GPS Heading
Vd_sat = u(13);                                 %GPS LTP Down Velocity
                                                % (Only used in Condor
                                                % Simulation)

% Process and Measurement Noise Matrices
R = diag(u(14:16));                             %AHRS Measurement Noise
R2 = diag(u(17:22));                            %INS Measurement Noise

Qv_C = diag(u(23:25));                          %AHRS Process Noise
Condor
Qv_S = diag(u(23:28));                          %AHRS Process Noise
SEAFOX
Qv2_C = diag(u(29:31));                         %INS Process Noise
Condor
Qv2_S = diag(u(29:34));                         %INS Process Noise
SEAFOX

% GPS Accelerations
ax_gps = u(35); ay_gps = u(36); az_gps = u(37); %GPS Accelerations as
                                                % calculated from 3rd
                                                % Order Filter

SIM = u(38);                                    %Condor(1) SEAFOX(2)
%% Initialization
persistent x_hat x_hat2 P P2 g dt ii jj H1 H2 H3 H4 Heading0 psi0 ...
           n1 n2 wei lat0 kk

% Allows time for Condor To steady out during live testing
if isempty(kk)
    kk = 0;
end
if kk < 1 && SIM==1
    X_HAT = zeros(16,1);
else

if isempty(x_hat)
% Miscellaneous
dt = 0.01;                                      %Filter dt [sec]
g = 9.8;                                        %Gravity Constant
[m/s^2]
n1 = 7;                                         %Number of AHRS States
n2 = 9;                                         %Number of INS States
wei = 7.292115*10^-5;                           %Sidereal Rate rad/s
lat0= 0.639268394832413; %lon0 = -2.115435878466264 [rad]

% AHRS Initialization
% Euler Angle Initialization
rx = ax_gps*cos(heading)+ay_gps*sin(heading);
ry = -ax_gps*sin(heading)+ay_gps*cos(heading);
rz = az_gps-g;
theta_0 = atan((-rx*rz - fx*sqrt(rx^2+rz^2-fx^2)) / (fx^2-rz^2));
r_theta = rx*sin(theta_0) + rz*cos(theta_0);
fc = fy-speed*r;
```

```
phi_0 = atan((r_theta*ry + fc*sqrt(ry^2+r_theta^2-fc^2)) / (fc^2-
r_theta^2));
psi_0=heading;
% Euler Angles to Quaternions
q_0 = e2q(phi_0, theta_0, psi_0);
% Gyro Bias Initial Values
bg_0 = [0; 0; 0];
% State Vector
x_hat = [q_0; bg_0];
% Initial Covariance Estimate
P = diag([1e-5*ones(1,4) 1e-3*ones(1,3)]);


%INS Initialization
% Position & Velocity Initialization
x_0 = N_sat;
y_0 = E_sat;
z_0 = D_sat;
vn_0 = speed*cos(CoG);
ve_0 = speed*sin(CoG);
vd_0 = Vd_sat;
% Accelerometer Initial Bias Estimate
ba_0 = [0; 0; 0];
% State Vector
x_hat2 = [x_0; y_0; z_0; vn_0; ve_0; vd_0; ba_0];
% Initial Covariance Estimate
P2 = diag([1e-5*ones(1,3), 1e-5*ones(1,3), 1e-3*ones(1,3)]);
end

%% AHRS ESTIMATOR

% Measurement Process Jacobian
HH1 = AHRS_H(x_hat);

% Kalman Gain Calculation
K1 = P*HH1'/(HH1*P*HH1' + R);

% Measurement Processing
rx = ax_gps*cos(heading)+ay_gps*sin(heading);
ry = -ax_gps*sin(heading)+ay_gps*cos(heading);
rz = az_gps-g;
% Theta
theta = atan((-rx*rz - fx*sqrt(rx^2+rz^2-fx^2)) / (fx^2-rz^2));
r_theta = rx*sin(theta) + rz*cos(theta);
% Added Coriolis Term
fc = fy-(norm(x_hat2(4:6)))*r;
% Phi
phi = atan((r_theta*ry + fc*sqrt(ry^2+r_theta^2-fc^2)) / (fc^2-
r_theta^2));
% Heading (Remove [0 2pi] restriction)
if isempty(H1)
    H1 = heading;
    H2 = 0;
    Heading0 = H1;
```

```
    Heading = H1;
    ii=0;
else
    H2 = heading;
    if (H2-H1) <= -100*pi/180
        ii = ii+1;
    end
    if (H2-H1) >= 100*pi/180
        ii = ii-1;
    end
    Heading = Heading0+((H2+2*pi*ii) - Heading0);
    H1 = H2;
end
% Measurement Vector
Z = [phi; theta; Heading];

% Measurement Estimate
[phi_hat, theta_hat, psi_hat] = q2e(x_hat);
% Heading (Remove [0 2pi] restriction)
if isempty(H3)
    H3 = psi_hat;
    H4 = 0;
    psi0 = H3;
    psi = H3;
    jj=0;
else
    H4 = psi_hat;
    if (H4-H3) <= -100*pi/180
        jj = jj+1;
    end
    if (H4-H3) >= 100*pi/180
        jj = jj-1;
    end
    psi = psi0+((H4+2*pi*jj) - psi0);
    H3 = H4;
end
% Measurment Estimate Vecotr
Z_hat =  [phi_hat; theta_hat; psi];

% Measurement Update
x_hat = x_hat + K1*(Z-Z_hat);
% Normalize Quaternions
Q = x_hat(1:4);
Qn = Q/sqrt(Q'*Q);
x_hat = [Qn; x_hat(5:7)];

% AHRS State Output at time k
X_AHRS = x_hat;

% Covariance Update
P = (eye(n1,n1)-K1*HH1)*P;

% Time Projection
if SIM == 1
```

```
    F = AHRS_F_condor(x_hat,p,q,r);
    G = AHRS_G_condor(x_hat);
    Qv = Qv_C;
else
    F = AHRS_F_SEAFOX(x_hat,p,q,r,wei,lat0);
    G = AHRS_G_SEAFOX(x_hat);
    Qv = Qv_S;
end

% Discretization of F and G
OMEGA = [-F G*Qv*G';
         zeros(n1,n1) F'];
GAMMA = expm(OMEGA*dt);
PHI = transpose(GAMMA((n1+1:2*n1),(n1+1):2*n1));
Qd = PHI*GAMMA((1:n1),(n1+1):2*n1);

% Covariance Time Projection
P = PHI*P*PHI' + Qd;

% State Time Projection
    w_bi = [p; q; r]-x_hat(5:7);
if SIM == 1
    w_bt = w_bi;
else
    R_t2b = rot_t2b(x_hat);
    w_bt = w_bi - R_t2b*[wei*cos(lat0); 0; -wei*sin(lat0)];
end

w1 = w_bt(1); w2 = w_bt(2); w3 = w_bt(3);
Q_kp1 = x_hat(1:4) + (dt/2)*[0 -w1 -w2 -w3;
                             w1  0  w3 -w2;
                             w2 -w3  0  w1;
                             w3  w2 -w1  0]*x_hat(1:4);
% Normalize the Quaternion
Q_kp1_n = Q_kp1/sqrt(Q_kp1'*Q_kp1);
x_hat = [Q_kp1_n; x_hat(5:7)];

%% INS ESTIMATOR

% Measurement Process Jacobian
HH2 = [ 1, 0, 0, 0, 0, 0, 0, 0, 0;
      0, 1, 0, 0, 0, 0, 0, 0, 0;
      0, 0, 1, 0, 0, 0, 0, 0, 0;
      0, 0, 0, 1, 0, 0, 0, 0, 0;
      0, 0, 0, 0, 1, 0, 0, 0, 0;
      0, 0, 0, 0, 0, 1, 0, 0, 0];

% Kalman Gain Calculation
K2 = P2*HH2'/(HH2*P2*HH2'+R2);

% Measurement
Z2 = [N_sat; E_sat; D_sat; speed*cos(CoG); speed*sin(CoG); Vd_sat];

% Measurement Estimate
```

```
Z2_hat = x_hat2(1:6);

% Measurement State Update
x_hat2 = x_hat2 + K2*(Z2-Z2_hat);
X_INS = x_hat2;

% Covariance Update
P2 = (eye(n2,n2)-K2*HH2)*P2;

if SIM == 1
    F2 = INS_F_condor(X_AHRS);
    G2 = INS_G_condor(X_AHRS);
    Qv2 = Qv2_C;
else
    F2 = INS_F_SEAFOX(X_AHRS, wei, lat0);
    G2 = INS_G_SEAFOX(X_AHRS);
    Qv2 = Qv2_S;
end

% Discretization of F and G
OMEGA2 = [-F2 G2*Qv2*G2';
          zeros(n2,n2) F2'];
GAMMA2 = expm(OMEGA2*dt);
PHI2 = transpose(GAMMA2((n2+1:2*n2),(n2+1):2*n2));
Qd2 = PHI2*GAMMA2((1:n2),(n2+1):2*n2);

% Covariance Time Projection
P2 = PHI2*P2*PHI2' + Qd2;

% State Time Projection
R_t2b = rot_t2b(X_AHRS);
R_b2t = R_t2b';

P_kp1 = x_hat2(1:3) + [x_hat2(4:5); -x_hat2(6)]*dt;
V_kp1 = x_hat2(4:6) + (R_b2t*([fx; fy; fz] - x_hat2(7:9)) + [0; 0;
g])*dt;
x_hat2 = [P_kp1; V_kp1; x_hat2(7:9)];

X_HAT = [X_AHRS(1:4); X_INS(1:6); X_AHRS(5:7); X_INS(7:9)];

end
kk = kk+1;

function R_t2b = rot_t2b(x)
q0 = x(1); q1 = x(2); q2 = x(3); q3 = x(4);
R_t2b = [q0^2+q1^2-q2^2-q3^2 2*(q1*q2+q0*q3) 2*(q1*q3-q0*q2);
         2*(q2*q1-q0*q3) q0^2-q1^2+q2^2-q3^2 2*(q2*q3+q0*q1);
         2*(q3*q1+q0*q2) 2*(q3*q2-q0*q1) q0^2-q1^2-q2^2+q3^2];

end

function G = INS_G_condor(x)
q0 = x(1); q1 = x(2); q2 = x(3); q3 = x(4);
```

```
G = [0, 0, 0;
     0, 0, 0;
     0, 0, 0;
     - q0^2 - q1^2 + q2^2 + q3^2, 2*q0*q3 - 2*q1*q2, - 2*q0*q2 -
2*q1*q3;
     - 2*q0*q3 - 2*q1*q2, - q0^2 + q1^2 - q2^2 + q3^2, 2*q0*q1 -
2*q2*q3;
     2*q0*q2 - 2*q1*q3, - 2*q0*q1 - 2*q2*q3, - q0^2 + q1^2 + q2^2 -
q3^2;
     0, 0, 0;
     0, 0, 0;
     0, 0, 0];

end

function G = INS_G_SEAFOX(x)
q0 = x(1); q1 = x(2); q2 = x(3); q3 = x(4);

G = [0, 0, 0, 0, 0, 0;
     0, 0, 0, 0, 0, 0;
     0, 0, 0, 0, 0, 0;
    -q0^2 - q1^2 + q2^2 + q3^2, 2*q0*q3 - 2*q1*q2, - 2*q0*q2 - ...
     2*q1*q3, 0, 0, 0;
    -2*q0*q3 - 2*q1*q2, - q0^2 + q1^2 - q2^2 + q3^2, 2*q0*q1 - ...
     2*q2*q3, 0, 0, 0;
     2*q0*q2 - 2*q1*q3, - 2*q0*q1 - 2*q2*q3, - q0^2 + q1^2 + q2^2 - ...
     q3^2, 0, 0, 0;
     0, 0, 0, 1, 0, 0;
     0, 0, 0, 0, 1, 0;
     0, 0, 0, 0, 0, 1];
end

function F = INS_F_condor(x)
q0 = x(1); q1 = x(2); q2 = x(3); q3 = x(4);
F = [0, 0, 0, 1, 0, 0, 0, 0, 0;
     0, 0, 0, 0, 1, 0, 0, 0, 0;
     0, 0, 0, 0, 0, -1, 0, 0, 0;
     0, 0, 0, 0, 0, 0, - q0^2 - q1^2 + q2^2 + q3^2, 2*q0*q3 - 2*q1*q2,
...
     - 2*q0*q2 - 2*q1*q3;
     0, 0, 0, 0, 0, 0, - 2*q0*q3 - 2*q1*q2, - q0^2 + q1^2 - ...
     q2^2 + q3^2, 2*q0*q1 - 2*q2*q3;
     0, 0, 0, 0, 0, 0, 2*q0*q2 - 2*q1*q3, ...
     - 2*q0*q1 - 2*q2*q3, - q0^2 + q1^2 + q2^2 - q3^2;
     0, 0, 0, 0, 0, 0, 0, 0, 0;
     0, 0, 0, 0, 0, 0, 0, 0, 0;
     0, 0, 0, 0, 0, 0, 0, 0, 0];
end

function F = INS_F_SEAFOX(x, wei, lat0)
q0 = x(1); q1 = x(2); q2 = x(3); q3 = x(4);
F = [0, 0, 0, 1, 0, 0, 0, 0, 0;
     0, 0, 0, 0, 1, 0, 0, 0, 0;
     0, 0, 0, 0, 0, -1, 0, 0, 0;
```

```
     0, 0, 0, 0, -2*wei*sin(lat0), 0, - q0^2 - q1^2 + q2^2 + q3^2, ...
     2*q0*q3 - 2*q1*q2,- 2*q0*q2 - 2*q1*q3;
     0, 0, 0, 2*wei*sin(lat0), 0, 2*wei*cos(lat0), - 2*q0*q3 - 2*q1*q2,
...
     - q0^2 + q1^2 - q2^2 + q3^2, 2*q0*q1 - 2*q2*q3;
     0, 0, 0, 0, -2*wei*cos(lat0), 0, 2*q0*q2 - 2*q1*q3, - 2*q0*q1 ...
     - 2*q2*q3, - q0^2 + q1^2 + q2^2 - q3^2;
     0, 0, 0, 0, 0, 0, 0, 0, 0;
     0, 0, 0, 0, 0, 0, 0, 0, 0;
     0, 0, 0, 0, 0, 0, 0, 0, 0];
end


function G = AHRS_G_SEAFOX(x)
q0 = x(1); q1 = x(2); q2 = x(3); q3 = x(4);

G = [  q1/2,  q2/2,  q3/2, 0, 0, 0;
      -q0/2,  q3/2, -q2/2, 0, 0, 0;
      -q3/2, -q0/2,  q1/2, 0, 0, 0;
       q2/2, -q1/2, -q0/2, 0, 0, 0;
          0,     0,     0, 1, 0, 0;
          0,     0,     0, 0, 1, 0;
          0,     0,     0, 0, 0, 1];

end

function G = AHRS_G_condor(x)

q0 = x(1); q1 = x(2); q2 = x(3); q3 = x(4);
G = [  q1/2,  q2/2,  q3/2;
      -q0/2,  q3/2, -q2/2;
      -q3/2, -q0/2,  q1/2;
       q2/2, -q1/2, -q0/2;
          0,     0,     0;
          0,     0,     0;
          0,     0,     0];
end

function F = AHRS_F_condor(x, p, q, r)
% Quaternions
q0 = x(1); q1 = x(2); q2 = x(3); q3 = x(4);
% Gyro Bias
bg1 = x(5); bg2 = x(6); bg3 = x(7);

F = [0, bg1/2 - p/2, bg2/2 - q/2, bg3/2 - r/2,  q1/2,  q2/2,  q3/2;
     p/2 - bg1/2,         0, r/2 - bg3/2, bg2/2 - q/2, -q0/2,  q3/2, -
q2/2;
     q/2 - bg2/2, bg3/2 - r/2,         0, p/2 - bg1/2, -q3/2, -q0/2,
q1/2;
     r/2 - bg3/2, q/2 - bg2/2, bg1/2 - p/2,         0,  q2/2, -q1/2, -
q0/2;
               0,           0,           0,           0,   0,     0,
0;
```

```
                0,           0,           0,           0,     0,   0,
0;
                0,           0,           0,           0,     0,   0,
0];
end

function F = AHRS_F_SEAFOX(x, p, q, r, wei, lat0)
% Quaternions
q0 = x(1); q1 = x(2); q2 = x(3); q3 = x(4);
% Gyro Bias
bg1 = x(5); bg2 = x(6); bg3 = x(7);

F = [q0*wei*(q1*cos(lat0) - q3*sin(lat0)), ...
(wei*cos(lat0)*q0^2)/2 + (3*wei*cos(lat0)*q1^2)/2 - ...
wei*sin(lat0)*q1*q3 + (wei*cos(lat0)*q2^2)/2 + ...
(wei*cos(lat0)*q3^2)/2 + bg1/2 - p/2, ...
bg2/2 - q/2 + q1*q2*wei*cos(lat0) - q2*q3*wei*sin(lat0), ...
bg3/2 - r/2 - (q0^2*wei*sin(lat0))/2 - (q1^2*wei*sin(lat0))/2 - ...
(q2^2*wei*sin(lat0))/2 - (3*q3^2*wei*sin(lat0))/2 +
q1*q3*wei*cos(lat0), ...
q1/2,  q2/2,  q3/2;
p/2 - bg1/2 - (3*q0^2*wei*cos(lat0))/2 - (q1^2*wei*cos(lat0))/2 - ...
(q2^2*wei*cos(lat0))/2 - (q3^2*wei*cos(lat0))/2 - q0*q2*wei*sin(lat0),
...
-q1*wei*(q0*cos(lat0) + q2*sin(lat0)), r/2 - bg3/2 - ...
(q0^2*wei*sin(lat0))/2 - (q1^2*wei*sin(lat0))/2 - ...
(3*q2^2*wei*sin(lat0))/2 - (q3^2*wei*sin(lat0))/2 - ...
q0*q2*wei*cos(lat0), bg2/2 - q/2 - q0*q3*wei*cos(lat0) - ...
q2*q3*wei*sin(lat0), -q0/2,  q3/2, -q2/2;
q/2 - bg2/2 + q0*q3*wei*cos(lat0) + q0*q1*wei*sin(lat0), ...
(wei*sin(lat0)*q0^2)/2 + (3*wei*sin(lat0)*q1^2)/2 + ...
wei*cos(lat0)*q1*q3 + (wei*sin(lat0)*q2^2)/2 + ...
(wei*sin(lat0)*q3^2)/2 + bg3/2 - r/2, ...
q2*wei*(q3*cos(lat0) + q1*sin(lat0)), ...
(wei*cos(lat0)*q0^2)/2 + (wei*cos(lat0)*q1^2)/2 + wei*sin(lat0)*q1*q3 +
...
(wei*cos(lat0)*q2^2)/2 + (3*wei*cos(lat0)*q3^2)/2 - bg1/2 + ...
p/2, -q3/2, -q0/2,  q1/2;
(3*wei*sin(lat0)*q0^2)/2 - wei*cos(lat0)*q0*q2 + ...
(wei*sin(lat0)*q1^2)/2 + (wei*sin(lat0)*q2^2)/2 + ...
(wei*sin(lat0)*q3^2)/2 - bg3/2 + r/2, q/2 - ...
bg2/2 - q1*q2*wei*cos(lat0) + q0*q1*wei*sin(lat0), ...
bg1/2 - p/2 - (q0^2*wei*cos(lat0))/2 - (q1^2*wei*cos(lat0))/2 - ...
(3*q2^2*wei*cos(lat0))/2 - (q3^2*wei*cos(lat0))/2 + ...
q0*q2*wei*sin(lat0), -q3*wei*(q2*cos(lat0) - q0*sin(lat0)),  ...
q2/2, -q1/2, -q0/2;
0, 0, 0, 0, 0, 0, 0;
0, 0, 0, 0, 0, 0, 0;
0, 0, 0, 0, 0, 0, 0];
end

function H = AHRS_H(x)
q0 = x(1); q1 = x(2); q2 = x(3); q3 = x(4);

H = [ -(2*(q0^2*q1 + 2*q0*q2*q3 + q1^3 + q1*q2^2 - q1*q3^2))/ ...
```

```
(((4*(q0*q1 + q2*q3)^2)/(q0^2 - q1^2 - q2^2 + q3^2)^2 + 1)* ...
(q0^2 - q1^2 - q2^2 + q3^2)^2),    (2*(q0^3 + q0*q1^2 - q0*q2^2 + ...
q0*q3^2 + 2*q1*q2*q3))/(((4*(q0*q1 + q2*q3)^2)/(q0^2 - q1^2 - ...
q2^2 + q3^2)^2 + 1)*(q0^2 - q1^2 - q2^2 + q3^2)^2), (2*(q0^2*q3 + ...
2*q0*q1*q2 - q1^2*q3 + q2^2*q3 + q3^3))/(((4*(q0*q1 + q2*q3)^2)/ ...
(q0^2 - q1^2 - q2^2 + q3^2)^2 + 1)*(q0^2 - q1^2 - q2^2 + q3^2)^2), ...
-(2*(- q0^2*q2 + 2*q0*q1*q3 + q1^2*q2 + q2^3 + q2*q3^2))/(((4* ...
(q0*q1 + q2*q3)^2)/(q0^2 - q1^2 - q2^2 + q3^2)^2 + 1)*(q0^2 - q1^2 ...
- q2^2 + q3^2)^2), 0, 0, 0;
(2*q2)/(1 - (2*q0*q2 - 2*q1*q3)^2)^(1/2), -(2*q3)/(1 - (2*q0*q2 - ...
2*q1*q3)^2)^(1/2), (2*q0)/(1 - (2*q0*q2 - 2*q1*q3)^2)^(1/2), ...
-(2*q1)/(1 - (2*q0*q2 - 2*q1*q3)^2)^(1/2), 0, 0, 0;
-(2*(q0^2*q3 + 2*q0*q1*q2 - q1^2*q3 + q2^2*q3 + q3^3))/(((4*(q0*q3 ...
+ q1*q2)^2)/(q0^2 + q1^2 - q2^2 - q3^2)^2 + 1)*(q0^2 + q1^2 - q2^2 ...
- q3^2)^2), -(2*(- q0^2*q2 + 2*q0*q1*q3 + q1^2*q2 + q2^3 + ...
q2*q3^2))/(((4*(q0*q3 + q1*q2)^2)/(q0^2 + q1^2 - q2^2 - q3^2)^2 + ...
1)*(q0^2 + q1^2 - q2^2 - q3^2)^2), (2*(q0^2*q1 + 2*q0*q2*q3 + q1^3 +
...
q1*q2^2 - q1*q3^2))/(((4*(q0*q3 + q1*q2)^2)/(q0^2 + q1^2 - q2^2 - ...
q3^2)^2 + 1)*(q0^2 + q1^2 - q2^2 - q3^2)^2),    (2*(q0^3 + q0*q1^2 ...
- q0*q2^2 + q0*q3^2 + 2*q1*q2*q3))/(((4*(q0*q3 + q1*q2)^2)/(q0^2 + ...
q1^2 - q2^2 - q3^2)^2 + 1)*(q0^2 + q1^2 - q2^2 - q3^2)^2), 0, 0, 0];
end

function q = e2q(phi, theta, psi)
% Form Rotation Matrix
R_psi = [cos(psi) sin(psi) 0; -sin(psi) cos(psi) 0; 0 0 1];
R_theta = [cos(theta) 0 -sin(theta); 0 1 0; sin(theta) 0 cos(theta)];
R_phi = [1 0 0; 0 cos(phi) sin(phi); 0 -sin(phi) cos(phi)];
R_n2b = R_phi*R_theta*R_psi;

% Extract Quaternions
q0=sqrt(1+trace(R_n2b))/2;
q1=(R_n2b(2,3)-R_n2b(3,2))/(4*q0);
q2=(R_n2b(3,1)-R_n2b(1,3))/(4*q0);
q3=(R_n2b(1,2)-R_n2b(2,1))/(4*q0);

q = [q0; q1; q2; q3];
end

function [phi, theta, psi] = q2e(x)
q0 = x(1); q1 = x(2); q2 = x(3); q3 = x(4);

% Compute Euler Angles
phi = atan2(2*(q2*q3+q0*q1),(q0^2-q1^2-q2^2+q3^2));
theta =  asin(-2*(q1*q3-q0*q2));
psi = mod(atan2(2*(q1*q2+q0*q3),(q0^2+q1^2-q2^2-q3^2)),2*pi);
end

end
```

2. Measurement Model Two Implementation

```
function X_HAT = EKF2(u)
```

```
%% Extended Kalman Filter for Navigation State Estimation
% Measurement Model 2
% LT Steven Terjesen
% September 2014

% ** This EKF was unable to converge when running measurement model 2
**

% This estimator is built for use in MATLAB Simulink. There are 39
inputs
% required and can be run in Simulink with an "Interpolated MATLAB
% Function" block. The first 13 inputs are the vehicle data: Course
Over
% Ground, Speed Over Ground, Accelerometer Measurements, Gyro
Measurements,
% GPS Measurements (In LTP XNorth-YEast-ZDown), Heading Measurement,
and
% Vertical Velocity (zeroed out for SEAFOXII data). Inputs 14 through
35
% are the Process Noise and Measurement Noise diagonal elements. These
% elements are not hard coded to allow for easier tuning. Inputs 36
% through 38 are the N-E-D accelerations in the LTP frame. Input 39 is
a
% toggle for selecting whether the data is from Condor or SEAFOX II.

%% System Inputs
% Measurement inputs
CoG = u(1);                                          %GPS Course Over Ground
speed = u(2);                                        %GPS Speed Over Ground
fx = u(3); fy = u(4); fz = u(5);                     %IMU Accelerations
p = u(6); q = u(7); r = u(8);                        %IMU Angular Rates
N_sat = u(9); E_sat = u(10); D_sat = u(11);          %GPS Position (NED)
heading = u(12);                                     %GPS Heading
Vd_sat = u(13);                                      %GPS LTP Down Velocity
                                                     % (Only used in Condor
                                                     % Simulation)
% Process and Measurement Noise Matrices
R = diag(u(14:17));                                  %AHRS Measurement Noise
R2 = diag(u(18:23));                                 %INS Measurement Noise

Qv_C = diag(u(24:26));                               %AHRS Process Noise
Condor
Qv_S = diag(u(24:29));                               %AHRS Process Noise
SEAFOX
Qv2_C = diag(u(30:32));                              %INS Process Noise
Condor
Qv2_S = diag(u(30:35));                              %INS Process Noise
SEAFOX

% GPS Accelerations
ax_gps = u(36); ay_gps = u(37); az_gps = u(38); %GPS Accelerations as
                                                     % calculated from 3rd
                                                     % Order Filter
```

```
SIM = u(39);                                        %Condor(1) SEAFOX(2)
%% Initialization
persistent x_hat x_hat2 P P2 g dt ii jj H1 H2 H3 H4 Heading0 psi0 ...
           n1 n2 wei lat0 kk

% Allows time for Condor To steady out during live testing
if isempty(kk)
    kk = 0;
end
if kk < 1 && SIM==1
    X_HAT = zeros(16,1);
else

if isempty(x_hat)
% Miscellaneous
dt = 0.01;                                          %Filter dt [sec]
g = 9.8;                                            %Gravity Constant
[m/s^2]
n1 = 7;                                             %Number of AHRS States
n2 = 9;                                             %Number of INS States
wei = 7.292115*10^-5;                               %Sidereal Rate rad/s
lat0= 0.639268394832413; %lon0 = -2.115435878466264 [rad]

% AHRS Initialization
% Euler Angle Initialization
rx = ax_gps*cos(heading)+ay_gps*sin(heading);
ry = -ax_gps*sin(heading)+ay_gps*cos(heading);
rz = az_gps-g;
theta_0 = atan((-rx*rz - fx*sqrt(rx^2+rz^2-fx^2)) / (fx^2-rz^2));
r_theta = rx*sin(theta_0) + rz*cos(theta_0);
fc = fy-speed*r;
phi_0 = atan((r_theta*ry + fc*sqrt(ry^2+r_theta^2-fc^2)) / (fc^2-
r_theta^2));
psi_0=heading;
% Euler Angles to Quaternions
q_0 = e2q(phi_0, theta_0, psi_0);
% Gyro Bias Initial Values
bg_0 = [0; 0; 0];
% State Vector
x_hat = [q_0; bg_0];
% Initial Covariance Estimate
P = diag([1e-3*ones(1,4) 1e-3*ones(1,3)]);

%INS Initialization
% Position & Velocity Initialization
x_0 = N_sat;
y_0 = E_sat;
z_0 = D_sat;
vn_0 = speed*cos(CoG);
ve_0 = speed*sin(CoG);
vd_0 = Vd_sat;
% Accelerometer Initial Bias Estimate
ba_0 = [0; 0; 0];
% State Vector
x_hat2 = [x_0; y_0; z_0; vn_0; ve_0; vd_0; ba_0];
```

```
% Initial Covariance Estimate
P2 = diag([1e-3*ones(1,3), 1e-3*ones(1,3), 1e-3*ones(1,3)]);
end

%% AHRS ESTIMATOR

% Measurement Process Jacobian
H = AHRS_H(x_hat, x_hat2, p, q, r, g, ax_gps, ay_gps, az_gps);

% Kalman Gain Calculation
K = P*H'/(H*P*H' + R);

% Measurement Processing
% Heading (Remove [0 2pi] restriction)
if isempty(H1)
    H1 = heading;
    H2 = 0;
    Heading0 = H1;
    Heading = H1;
    ii=0;
else
    H2 = heading;
    if (H2-H1) <= -100*pi/180
        ii = ii+1;
    end
    if (H2-H1) >= 100*pi/180
        ii = ii-1;
    end
    Heading = Heading0+((H2+2*pi*ii) - Heading0);
    H1 = H2;
end
% Measurement Vector
Z = [fx; fy; fz; Heading];

% Measurement Estimate
[~, ~, psi_hat] = q2e(x_hat);
% Heading (Remove [0 2pi] restriction)
if isempty(H3)
    H3 = psi_hat;
    H4 = 0;
    psi0 = H3;
    psi = H3;
    jj=0;
else
    H4 = psi_hat;
    if (H4-H3) <= -100*pi/180
        jj = jj+1;
    end
    if (H4-H3) >= 100*pi/180
        jj = jj-1;
    end
    psi = psi0+((H4+2*pi*jj) - psi0);
    H3 = H4;
end
```

```
R_t2b = rot_t2b(x_hat);
w_bi = [p; q; r] - x_hat(5:7);
if SIM == 1
    w_bt = w_bi;
else
    w_bt = w_bi - R_t2b*[wei*cos(lat0); 0; -wei*sin(lat0)];
end

Fb_hat = R_t2b*[ax_gps; ay_gps; az_gps] + cross(w_bt,
R_t2b*x_hat2(4:6)) + x_hat2(7:9);

% Measurment Estimate Vecotr
Z_hat =  [Fb_hat; psi];

% Measurement Update
x_hat = x_hat + K*(Z-Z_hat);
% Normalize Quaternions
Q = x_hat(1:4);
Qn = Q/sqrt(Q'*Q);
x_hat = [Qn; x_hat(5:7)];

% AHRS State Output at time k
X_AHRS = x_hat;

% Covariance Update
P = (eye(n1,n1)-K*H)*P;

% Time Projection
if SIM == 1
    F = AHRS_F_condor(x_hat,p,q,r);
    G = AHRS_G_condor(x_hat);
    Qv = Qv_C;
else
    F = AHRS_F_SEAFOX(x_hat,p,q,r,wei,lat0);
    G = AHRS_G_SEAFOX(x_hat);
    Qv = Qv_S;
end

% Discretization of F and G
OMEGA = [-F G*Qv*G';
         zeros(n1,n1) F'];
GAMMA = expm(OMEGA*dt);
PHI = transpose(GAMMA((n1+1:2*n1),(n1+1):2*n1));
Qd = PHI*GAMMA((1:n1),(n1+1):2*n1);

% Covariance Time Projection
P = PHI*P*PHI' + Qd;

% State Time Projection
    w_bi = [p; q; r]-x_hat(5:7);
if SIM == 1
    w_bt = w_bi;
```

```
else
    R_t2b = rot_t2b(x_hat);
    w_bt = w_bi - R_t2b*[wei*cos(lat0); 0; -wei*sin(lat0)];
end

w1 = w_bt(1); w2 = w_bt(2); w3 = w_bt(3);
Q_kp1 = x_hat(1:4) + (dt/2)*[0 -w1 -w2 -w3;
                             w1  0  w3 -w2;
                             w2 -w3  0  w1;
                             w3  w2 -w1  0]*x_hat(1:4);
% Normalize the Quaternion
Q_kp1_n = Q_kp1/sqrt(Q_kp1'*Q_kp1);
x_hat = [Q_kp1_n; x_hat(5:7)];

%% INS ESTIMATOR

% Measurement Process Jacobian
H = [ 1, 0, 0, 0, 0, 0, 0, 0, 0;
      0, 1, 0, 0, 0, 0, 0, 0, 0;
      0, 0, 1, 0, 0, 0, 0, 0, 0;
      0, 0, 0, 1, 0, 0, 0, 0, 0;
      0, 0, 0, 0, 1, 0, 0, 0, 0;
      0, 0, 0, 0, 0, 1, 0, 0, 0];

% Kalman Gain Calculation
K = P2*H'/(H*P2*H'+R2);

% Measurement
Z = [N_sat; E_sat; D_sat; speed*cos(CoG); speed*sin(CoG); Vd_sat];

% Measurement Estimate
Z_hat = x_hat2(1:6);

% Measurement State Update
x_hat2 = x_hat2 + K*(Z-Z_hat);
X_INS = x_hat2;

% Covariance Update
P2 = (eye(n2,n2)-K*H)*P2;

if SIM == 1
    F = INS_F_condor(X_AHRS);
    G = INS_G_condor(X_AHRS);
    Qv2 = Qv2_C;
else
    F = INS_F_SEAFOX(X_AHRS, wei, lat0);
    G = INS_G_SEAFOX(X_AHRS);
    Qv2 = Qv2_S;
end

% Discretization of F and G
OMEGA = [-F G*Qv2*G';
         zeros(n2,n2) F'];
```

```
GAMMA = expm(OMEGA*dt);
PHI = transpose(GAMMA((n2+1:2*n2),(n2+1):2*n2));
Qd = PHI*GAMMA((1:n2),(n2+1):2*n2);

% Covariance Time Projection
P2 = PHI*P2*PHI' + Qd;

% State Time Projection
R_t2b = rot_t2b(X_AHRS);
R_b2t = R_t2b';

P_kp1 = x_hat2(1:3) + [x_hat2(4:5); -x_hat(6)]*dt;
V_kp1 = x_hat2(4:6) + (R_b2t*([fx; fy; fz] - x_hat2(7:9)) + [0; 0;
g])*dt;
x_hat2 = [P_kp1; V_kp1; x_hat2(7:9)];

X_HAT = [X_AHRS; X_INS];
end
kk = kk+1;

function R_t2b = rot_t2b(x)
q0 = x(1); q1 = x(2); q2 = x(3); q3 = x(4);
R_t2b = [q0^2+q1^2-q2^2-q3^2 2*(q1*q2+q0*q3) 2*(q1*q3-q0*q2);
         2*(q2*q1-q0*q3) q0^2-q1^2+q2^2-q3^2 2*(q2*q3+q0*q1);
         2*(q3*q1+q0*q2) 2*(q3*q2-q0*q1) q0^2-q1^2-q2^2+q3^2];

end

function G = INS_G_condor(x)
q0 = x(1); q1 = x(2); q2 = x(3); q3 = x(4);
G = [0, 0, 0;
     0, 0, 0;
     0, 0, 0;
     - q0^2 - q1^2 + q2^2 + q3^2, 2*q0*q3 - 2*q1*q2, - 2*q0*q2 -
2*q1*q3;
     - 2*q0*q3 - 2*q1*q2, - q0^2 + q1^2 - q2^2 + q3^2, 2*q0*q1 -
2*q2*q3;
     2*q0*q2 - 2*q1*q3, - 2*q0*q1 - 2*q2*q3, - q0^2 + q1^2 + q2^2 -
q3^2;
     0, 0, 0;
     0, 0, 0;
     0, 0, 0];

end

function G = INS_G_SEAFOX(x)
q0 = x(1); q1 = x(2); q2 = x(3); q3 = x(4);

G = [0, 0, 0, 0, 0, 0;
     0, 0, 0, 0, 0, 0;
     0, 0, 0, 0, 0, 0;
    -q0^2 - q1^2 + q2^2 + q3^2, 2*q0*q3 - 2*q1*q2, - 2*q0*q2 - ...
     2*q1*q3, 0, 0, 0;
```

```
    -2*q0*q3 - 2*q1*q2, - q0^2 + q1^2 - q2^2 + q3^2, 2*q0*q1 - ...
     2*q2*q3, 0, 0, 0;
     2*q0*q2 - 2*q1*q3, - 2*q0*q1 - 2*q2*q3, - q0^2 + q1^2 + q2^2 - ...
     q3^2, 0, 0, 0;
     0, 0, 0, 1, 0, 0;
     0, 0, 0, 0, 1, 0;
     0, 0, 0, 0, 0, 1];
end

function F = INS_F_condor(x)
q0 = x(1); q1 = x(2); q2 = x(3); q3 = x(4);
F = [0, 0, 0, 1, 0, 0, 0, 0, 0;
     0, 0, 0, 0, 1, 0, 0, 0, 0;
     0, 0, 0, 0, 0, -1, 0, 0, 0;
     0, 0, 0, 0, 0, 0, - q0^2 - q1^2 + q2^2 + q3^2, 2*q0*q3 - 2*q1*q2,
...
     - 2*q0*q2 - 2*q1*q3;
     0, 0, 0, 0, 0, 0, - 2*q0*q3 - 2*q1*q2, - q0^2 + q1^2 - ...
     q2^2 + q3^2, 2*q0*q1 - 2*q2*q3;
     0, 0, 0, 0, 0, 0, 2*q0*q2 - 2*q1*q3, ...
     - 2*q0*q1 - 2*q2*q3, - q0^2 + q1^2 + q2^2 - q3^2;
     0, 0, 0, 0, 0, 0, 0, 0, 0;
     0, 0, 0, 0, 0, 0, 0, 0, 0;
     0, 0, 0, 0, 0, 0, 0, 0, 0];
end

function F = INS_F_SEAFOX(x, wei, lat0)
q0 = x(1); q1 = x(2); q2 = x(3); q3 = x(4);
F = [0, 0, 0, 1, 0, 0, 0, 0, 0;
     0, 0, 0, 0, 1, 0, 0, 0, 0;
     0, 0, 0, 0, 0, -1, 0, 0, 0;
     0, 0, 0, 0, -2*wei*sin(lat0), 0, - q0^2 - q1^2 + q2^2 + q3^2, ...
     2*q0*q3 - 2*q1*q2,- 2*q0*q2 - 2*q1*q3;
     0, 0, 0, 2*wei*sin(lat0), 0, 2*wei*cos(lat0), - 2*q0*q3 - 2*q1*q2,
...
     - q0^2 + q1^2 - q2^2 + q3^2, 2*q0*q1 - 2*q2*q3;
     0, 0, 0, 0, -2*wei*cos(lat0), 0, 2*q0*q2 - 2*q1*q3, - 2*q0*q1 ...
     - 2*q2*q3, - q0^2 + q1^2 + q2^2 - q3^2;
     0, 0, 0, 0, 0, 0, 0, 0, 0;
     0, 0, 0, 0, 0, 0, 0, 0, 0;
     0, 0, 0, 0, 0, 0, 0, 0, 0];
end


function G = AHRS_G_SEAFOX(x)
q0 = x(1); q1 = x(2); q2 = x(3); q3 = x(4);

G = [  q1/2,  q2/2,  q3/2, 0, 0, 0;
      -q0/2,  q3/2, -q2/2, 0, 0, 0;
      -q3/2, -q0/2,  q1/2, 0, 0, 0;
       q2/2, -q1/2, -q0/2, 0, 0, 0;
          0,     0,     0, 1, 0, 0;
          0,     0,     0, 0, 1, 0;
          0,     0,     0, 0, 0, 1];
```

```
end

function G = AHRS_G_condor(x)

q0 = x(1); q1 = x(2); q2 = x(3); q3 = x(4);
G = [  q1/2,  q2/2,  q3/2;
      -q0/2,  q3/2, -q2/2;
      -q3/2, -q0/2,  q1/2;
       q2/2, -q1/2, -q0/2;
          0,     0,     0;
          0,     0,     0;
          0,     0,     0];
end

function F = AHRS_F_condor(x, p, q, r)
% Quaternions
q0 = x(1); q1 = x(2); q2 = x(3); q3 = x(4);
% Gyro Bias
bg1 = x(5); bg2 = x(6); bg3 = x(7);

F = [0, bg1/2 - p/2, bg2/2 - q/2, bg3/2 - r/2,  q1/2,  q2/2,  q3/2;
     p/2 - bg1/2,         0, r/2 - bg3/2, bg2/2 - q/2, -q0/2,  q3/2, -
q2/2;
     q/2 - bg2/2, bg3/2 - r/2,         0, p/2 - bg1/2, -q3/2, -q0/2,
q1/2;
     r/2 - bg3/2, q/2 - bg2/2, bg1/2 - p/2,         0,  q2/2, -q1/2, -
q0/2;
               0,           0,           0,           0,   0,     0,
0;
               0,           0,           0,           0,     0,   0,
0;
               0,           0,           0,           0,     0,   0,
0];
end

function F = AHRS_F_SEAFOX(x, p, q, r, wei, lat0)
% Quaternions
q0 = x(1); q1 = x(2); q2 = x(3); q3 = x(4);
% Gyro Bias
bg1 = x(5); bg2 = x(6); bg3 = x(7);

F = [q0*wei*(q1*cos(lat0) - q3*sin(lat0)), ...
(wei*cos(lat0)*q0^2)/2 + (3*wei*cos(lat0)*q1^2)/2 - ...
wei*sin(lat0)*q1*q3 + (wei*cos(lat0)*q2^2)/2 + ...
(wei*cos(lat0)*q3^2)/2 + bg1/2 - p/2, ...
bg2/2 - q/2 + q1*q2*wei*cos(lat0) - q2*q3*wei*sin(lat0), ...
bg3/2 - r/2 - (q0^2*wei*sin(lat0))/2 - (q1^2*wei*sin(lat0))/2 - ...
(q2^2*wei*sin(lat0))/2 - (3*q3^2*wei*sin(lat0))/2 +
q1*q3*wei*cos(lat0), ...
q1/2,  q2/2,  q3/2;
p/2 - bg1/2 - (3*q0^2*wei*cos(lat0))/2 - (q1^2*wei*cos(lat0))/2 - ...
(q2^2*wei*cos(lat0))/2 - (q3^2*wei*cos(lat0))/2 - q0*q2*wei*sin(lat0),
...
```

```
-q1*wei*(q0*cos(lat0) + q2*sin(lat0)), r/2 - bg3/2 - ...
(q0^2*wei*sin(lat0))/2 - (q1^2*wei*sin(lat0))/2 - ...
(3*q2^2*wei*sin(lat0))/2 - (q3^2*wei*sin(lat0))/2 - ...
q0*q2*wei*cos(lat0), bg2/2 - q/2 - q0*q3*wei*cos(lat0) - ...
q2*q3*wei*sin(lat0), -q0/2,  q3/2, -q2/2;
q/2 - bg2/2 + q0*q3*wei*cos(lat0) + q0*q1*wei*sin(lat0), ...
(wei*sin(lat0)*q0^2)/2 + (3*wei*sin(lat0)*q1^2)/2 + ...
wei*cos(lat0)*q1*q3 + (wei*sin(lat0)*q2^2)/2 + ...
(wei*sin(lat0)*q3^2)/2 + bg3/2 - r/2, ...
q2*wei*(q3*cos(lat0) + q1*sin(lat0)), ...
(wei*cos(lat0)*q0^2)/2 + (wei*cos(lat0)*q1^2)/2 + wei*sin(lat0)*q1*q3 +
...
(wei*cos(lat0)*q2^2)/2 + (3*wei*cos(lat0)*q3^2)/2 - bg1/2 + ...
p/2, -q3/2, -q0/2,  q1/2;
(3*wei*sin(lat0)*q0^2)/2 - wei*cos(lat0)*q0*q2 + ...
(wei*sin(lat0)*q1^2)/2 + (wei*sin(lat0)*q2^2)/2 + ...
(wei*sin(lat0)*q3^2)/2 - bg3/2 + r/2, q/2 - ...
bg2/2 - q1*q2*wei*cos(lat0) + q0*q1*wei*sin(lat0), ...
bg1/2 - p/2 - (q0^2*wei*cos(lat0))/2 - (q1^2*wei*cos(lat0))/2 - ...
(3*q2^2*wei*cos(lat0))/2 - (q3^2*wei*cos(lat0))/2 + ...
q0*q2*wei*sin(lat0), -q3*wei*(q2*cos(lat0) - q0*sin(lat0)),  ...
q2/2, -q1/2, -q0/2;
0, 0, 0, 0, 0, 0, 0;
0, 0, 0, 0, 0, 0, 0;
0, 0, 0, 0, 0, 0, 0];
end

function H = AHRS_H(x, x2, p, q, r, g, ax, ay, az)
q0 = x(1); q1 = x(2); q2 = x(3); q3 = x(4);
bg1 = x(5); bg2 = x(6); bg3 = x(7);
vn = x2(4); ve = x2(5); vd = x2(6);

H = [(bg3 - r)*(2*q1*vd + 2*q0*ve - 2*q3*vn) - (bg2 - q)*(2*q0*vd - ...
    2*q1*ve + 2*q2*vn) + 2*ax*q0 + 2*ay*q3 - 2*az*q2 - 2*g*q2, ...
    (bg2 - q)*(2*q1*vd + 2*q0*ve - 2*q3*vn) + (bg3 - r)*(2*q0*vd - ...
    2*q1*ve + 2*q2*vn) + 2*ax*q1 + 2*ay*q2 + 2*az*q3 + 2*g*q3, ...
    (bg3 - r)*(2*q3*vd + 2*q2*ve + 2*q1*vn) - (bg2 - q)*(2*q3*ve - ...
    2*q2*vd + 2*q0*vn) - 2*ax*q2 + 2*ay*q1 - 2*az*q0 - 2*g*q0, ...
    2*ay*q0 - (bg3 - r)*(2*q3*ve - 2*q2*vd + 2*q0*vn) - 2*ax*q3 - ...
    (bg2 - q)*(2*q3*vd + 2*q2*ve + 2*q1*vn) + 2*az*q1 + 2*g*q1,
0, ve*(2*q0*q1 - 2*q2*q3) - vd*(q0^2 - q1^2 - q2^2 + q3^2) -
vn*(2*q0*q2 + 2*q1*q3), ve*(q0^2 - q1^2 + q2^2 - q3^2) + vd*(2*q0*q1 +
2*q2*q3) - vn*(2*q0*q3 - 2*q1*q2);
    (bg1 - p)*(2*q0*vd - 2*q1*ve + 2*q2*vn) - (bg3 - r)*(2*q3*ve - ...
    2*q2*vd + 2*q0*vn) - 2*ax*q3 + 2*ay*q0 + 2*az*q1 + 2*g*q1, ...
    2*ax*q2 - (bg3 - r)*(2*q3*vd + 2*q2*ve + 2*q1*vn) - (bg1 - ...
    p)*(2*q1*vd + 2*q0*ve - 2*q3*vn) - 2*ay*q1 + 2*az*q0 + 2*g*q0, ...
    (bg1 - p)*(2*q3*ve - 2*q2*vd + 2*q0*vn) + (bg3 - r)*(2*q0*vd - ...
    2*q1*ve + 2*q2*vn) + 2*ax*q1 + 2*ay*q2 + 2*az*q3 + 2*g*q3, ...
    (bg1 - p)*(2*q3*vd + 2*q2*ve + 2*q1*vn) - (bg3 - r)*(2*q1*vd + ...
    2*q0*ve - 2*q3*vn) - 2*ax*q0 - 2*ay*q3 + 2*az*q2 + 2*g*q2, ...
    vd*(q0^2 - q1^2 - q2^2 + q3^2) - ve*(2*q0*q1 - 2*q2*q3) + ...
    vn*(2*q0*q2 + 2*q1*q3), 0, vd*(2*q0*q2 - 2*q1*q3) - vn*(q0^2 + ...
    q1^2 - q2^2 - q3^2) - ve*(2*q0*q3 + 2*q1*q2);
    (bg2 - q)*(2*q3*ve - 2*q2*vd + 2*q0*vn) - (bg1 - p)*(2*q1*vd + ...
```

```
    2*q0*ve - 2*q3*vn) + 2*ax*q2 - 2*ay*q1 + 2*az*q0 + 2*g*q0, ...
    (bg2 - q)*(2*q3*vd + 2*q2*ve + 2*q1*vn) - (bg1 - p)*(2*q0*vd - ...
    2*q1*ve + 2*q2*vn) + 2*ax*q3 - 2*ay*q0 - 2*az*q1 - 2*g*q1, ...
    2*ax*q0 - (bg2 - q)*(2*q0*vd - 2*q1*ve + 2*q2*vn) - (bg1 - ...
    p)*(2*q3*vd + 2*q2*ve + 2*q1*vn) + 2*ay*q3 - 2*az*q2 - 2*g*q2, ...
    (bg1 - p)*(2*q3*ve - 2*q2*vd + 2*q0*vn) + (bg2 - q)*(2*q1*vd + ...
    2*q0*ve - 2*q3*vn) + 2*ax*q1 + 2*ay*q2 + 2*az*q3 + 2*g*q3, ...
    vn*(2*q0*q3 - 2*q1*q2) - vd*(2*q0*q1 + 2*q2*q3) - ve*(q0^2 - ...
    q1^2 + q2^2 - q3^2), vn*(q0^2 + q1^2 - q2^2 - q3^2) - vd*(2*q0*q2
...
    - 2*q1*q3) + ve*(2*q0*q3 + 2*q1*q2), 0;
   -(2*(q0^2*q3 + 2*q0*q1*q2 - q1^2*q3 + q2^2*q3 + q3^3))/(((4*(q0*q3 +
...
   q1*q2)^2)/(q0^2 + q1^2 - q2^2 - q3^2)^2 + 1)*(q0^2 + q1^2 - q2^2 -
...
   q3^2)^2), -(2*(- q0^2*q2 + 2*q0*q1*q3 + q1^2*q2 + q2^3 + ...
   q2*q3^2))/(((4*(q0*q3 + q1*q2)^2)/(q0^2 + q1^2 - q2^2 - q3^2)^2 +
...
   1)*(q0^2 + q1^2 - q2^2 - q3^2)^2), (2*(q0^2*q1 + 2*q0*q2*q3 + q1^3
...
   + q1*q2^2 - q1*q3^2))/(((4*(q0*q3 + q1*q2)^2)/(q0^2 + q1^2 - q2^2
...
   - q3^2)^2 + 1)*(q0^2 + q1^2 - q2^2 - q3^2)^2), (2*(q0^3 + q0*q1^2 -
...
   q0*q2^2 + q0*q3^2 + 2*q1*q2*q3))/(((4*(q0*q3 + q1*q2)^2)/(q0^2 + ...
   q1^2 - q2^2 - q3^2)^2 + 1)*(q0^2 + q1^2 - q2^2 - q3^2)^2), 0, 0, 0];
end

function q = e2q(phi, theta, psi)
% Form Rotation Matrix
R_psi = [cos(psi) sin(psi) 0; -sin(psi) cos(psi) 0; 0 0 1];
R_theta = [cos(theta) 0 -sin(theta); 0 1 0; sin(theta) 0 cos(theta)];
R_phi = [1 0 0; 0 cos(phi) sin(phi); 0 -sin(phi) cos(phi)];
R_n2b = R_phi*R_theta*R_psi;

% Extract Quaternions
q0=sqrt(1+trace(R_n2b))/2;
q1=(R_n2b(2,3)-R_n2b(3,2))/(4*q0);
q2=(R_n2b(3,1)-R_n2b(1,3))/(4*q0);
q3=(R_n2b(1,2)-R_n2b(2,1))/(4*q0);

q = [q0; q1; q2; q3];
end

function [phi, theta, psi] = q2e(x)
q0 = x(1); q1 = x(2); q2 = x(3); q3 = x(4);

% Compute Euler Angles
phi = atan2(2*(q2*q3+q0*q1),(q0^2-q1^2-q2^2+q3^2));
theta =  asin(-2*(q1*q3-q0*q2));
psi = mod(atan2(2*(q1*q2+q0*q3),(q0^2+q1^2-q2^2-q3^2)),2*pi);
end

end
```

C. STANDARD UNSCENTED KALMAN FILTER MATLAB CODE

1. Measurement Model One Implementation

```
function X_HAT = UKF1(u)
%% SPHERICAL SIMPLEX UNSCENTED KALMAN FILTER
% Measurement Model 1
% LT Steven Terjesen
% September 2014

% This estimator is built for use in MATLAB Simulink. There are 38
inputs
% required and can be run in Simulink with an “Interpolated MATLAB
% Function” block. The first 13 inputs are the vehicle data: Course
Over
% Ground, Speed Over Ground, Accelerometer Measurements, Gyro
Measurements,
% GPS Measurements (In LTP XNorth-YEast-ZDown), Heading Measurement,
and
% Vertical Velocity (zeroed out for SEAFOXII data). Inputs 14 through
34
% are the Process Noise and Measurement Noise diagonal elements. These
% elements are not hard coded to allow for easier tuning. Inputs 35
% through 37 are the N-E-D accelerations in the LTP frame. Input 38 is
a
% toggle for selecting whether the data is from Condor or SEAFOX II.

%% System Inputs
% Measurement inputs
CoG = u(1);                                          %GPS Course Over Ground
speed = u(2);                                        %GPS Speed Over Ground
fx = u(3); fy = u(4); fz = u(5);                     %IMU Accelerations
p = u(6); q = u(7); r = u(8);                        %IMU Angular Rates
N_sat = u(9); E_sat = u(10); D_sat = u(11);          %GPS Position (NED)
heading = u(12);                                     %GPS Heading
Vd_sat = u(13);                                      %GPS LTP Down Velocity

% Process and Measurement Noise Matrices
R = diag(u(14:16));                                  %AHRS Measurement Noise
R2 = diag(u(17:22));                                 %INS Measurement Noise

SIM = u(38);                                         %Condor(1) or
SEAFOXII(2)
%                                                     Simulation Selector

if SIM == 1
    Qv = diag(u(23:25));                             %AHRS Process Noise
Condor
    Qv2 = diag(u(29:31));                            %INS Process Noise
Condor
else
    Qv = diag(u(23:28));                             %AHRS Process Noise
SEAFOX
```

```
    Qv2 = diag(u(29:34));                          %INS Process Noise
SEAFOX
    Vd_sat = 0;                                    %No Vertical Velocity
%                                                   measurement available
on
%                                                   SEAFOX II, set Vd=0.
    D_sat = 0;                                     %No Altitude
Measurement
%                                                   available on SEAFOXII,
set
%                                                   D_sat = 0;
end

% GPS Accelerations
ax_gps = u(35); ay_gps = u(36); az_gps = u(37); %GPS Accelerations as
                                                % calculated from 3rd
                                                % Order Filter


%% Initialization
persistent x_bar Heading0 H1 H2 ii jj psi0 H3 H4 ...
Px Px2 kk dt g nn nx nxa nQ nR n wc wm x_bar2 nx2 nxa2 n2 nQ2 nR2 wc2
...
wm2 eta eta2 wei lat0

% Allows time for Condor To steady out during live testing
if isempty(kk)
    kk = 0;
end
if kk < 1 && SIM==1
    X_HAT = zeros(16,1);
else

if isempty(x_bar)
% Miscellaneous
dt = 0.01;                                         %Filter dt [sec]
g = 9.8;                                           %Gravity Constant
[m/s^2]
wei = 7.292115*10^-5;                              %Sidereal Rate [rad/s]
lat0= 0.639268394832413;                           %Origin of LTP [rad]
%lon0 = -2.115435878466264
nn=1;                                              %Heading Count
Initializer

% AHRS Initialization
% Euler Angle Initialization
rx = ax_gps*cos(heading)+ay_gps*sin(heading);
ry = -ax_gps*sin(heading)+ay_gps*cos(heading);
rz = az_gps-g;
theta_0 = atan((-rx*rz - fx*sqrt(rx^2+rz^2-fx^2))/(fx^2-rz^2));
r_theta = rx*sin(theta_0) + rz*cos(theta_0);
fc = fy-speed*r;
phi_0 = atan((r_theta*ry + fc*sqrt(ry^2+r_theta^2-fc^2))/(fc^2-
r_theta^2));
psi_0=heading;
```

```
% Euler Angles to Quaternions
q_0 = e2q(phi_0, theta_0, psi_0);
% Gyro Bias Initial Values
bg_0 = [0; 0; 0];
% Inital State Vector
x_bar = [q_0; bg_0];
% Initial Covariance Estimate
Px = diag([1e-5*ones(1,4) 1e-3*ones(1,3)]);

%INS Initialization
% Position & Velocity Initialization
x_0 = N_sat;
y_0 = E_sat;
z_0 = D_sat;
vn_0 = speed*cos(CoG);
ve_0 = speed*sin(CoG);
vd_0 = Vd_sat;
% Accelerometer Initial Bias Estimate
ba_0 = [0; 0; 0];
% Initial State Vector
x_bar2 = [x_0; y_0; z_0; vn_0; ve_0; vd_0; ba_0];
% Initial Covariance Estimate
Px2 = diag([1e-5*ones(1,3), 1e-5*ones(1,3), 1e-3*ones(1,3)]);

% AHRS SIGMA Weights Initialization
nx = length(x_bar);                          % Number of AHRS States
nQ = length(diag(Qv));                       % Process Noise States
nR = length(diag(R));                        % Measurement Noise
States
nxa = nx+nQ+nR;                              % Total Augmented
States
n = 2*nxa+1;                                   % Number of
Iterations
alpha = 1e-3;                                % Tunable Scaleing
Factor
beta = 2;                                    % beta = 2 for Gaussian
PDF
kappa = 0;                                   % Tunable Secondary
Scaling
%                                              Factor.
lamda = alpha^2*(nxa+kappa)-nxa;             % Weighting Factor
wm = (1/(2*(nxa+lamda)))*ones(n,1);          % Measurement Weights
wc = wm;                                     % Covariance Weights
wm(1) = lamda/(nxa+lamda);                   % Zeroth Measurement
Weight
wc(1) = wm(1) + (1-alpha^2+beta);            % Zeroth Covariance
Weight
eta = sqrt(nxa+lamda);                       % Covariance Weighting
%                                              Factor

% INS SIGMA Weights Initialization
nx2 = length(x_bar2);                        % Number of INS States
nQ2 = length(diag(Qv2));                     % Process Noise States
nR2 = length(diag(R2));                      % Measurement Noise
States
```

```
nxa2 = nx2+nQ2+nR2;                                 % Total Augmented
States
n2 = 2*nxa2+1;                                        % Total Iterations
alpha2 = 1e-3;                                      % Tunable, 0<alpha<1
beta2 = 2;                                          % beta = 2 for Guassian
PDF
kappa2 = 0;                                         % Tunable Secondary
Scaling
%                                                     Factor.
lamda2 = alpha2^2*(nxa2+kappa2)-nxa2;               % Weighting Factor
wm2 = (1/(2*(nxa2+lamda2)))*ones(n2,1);             % Measurement Weights
wc2 = wm2;                                          % Covariance Weights
wm2(1) = lamda2/(nxa2+lamda2);                      % Zeroth Measurement
Weight
wc2(1) = wm2(1) + (1-alpha2^2+beta2);               % Zeroth Covariance
Weight
eta2 = sqrt(nxa2+lamda2);                           % Covariance Weighting
%                                                     Factor

end
%% AHRS ESTIMATOR

% Build Augmented State Vector
X = zeros(nxa, n);
xa = [x_bar; zeros(nxa-nx,1)];

% Build Augmented Covariance Matrix
Pxa = [Px zeros(nx,nQ) zeros(nx,nR);
        zeros(nQ,nx) Qv zeros(nQ,nR);
        zeros(nR,nx) zeros(nR,nQ) R];

% Build 2n+1 Sigma Vectors
c = eta*Pxa;
for k = 1:nxa+1
    if k == 1
        X(:,k) = xa;
    else
        X(:,k) = xa + c(k-1,:)';
        X(:,k+nxa) = xa - c(k-1,:)';
    end
end

% Time Update
Xx = zeros(nx, n);
x_bar = zeros(nx, 1);
for k = 1:n

% Gyro Model  w_bi(true) = w_bi(measured) - bias - noise
w_bi_b = [p-X(5,k)-X(8,k); q-X(6,k)-X(9,k); r-X(7, k)-X(10,k)];

if SIM == 1
    % No Sidereal Rate
    w_bt = w_bi_b;
else
```

```
    % Sidereal Rate Included
    R_t2b = rot_t2b(X(1:4,k));
    w_bt = w_bi_b - R_t2b*[wei*cos(lat0); 0; -wei*sin(lat0)];
end

% Quaternion Update
w1 = w_bt(1); w2 = w_bt(2); w3 = w_bt(3);
Q_kp1 = X(1:4, k) + (dt/2)*[0,  -w1,  -w2,  -w3;
                        w1,  0,    w3,  -w2;
                        w2, -w3,   0,    w1;
                        w3,  w2,  -w1,   0]*X(1:4, k);

% Quaternion Normalization
Q_kp1_n = Q_kp1/sqrt(Q_kp1'*Q_kp1);

% Gyro Bias Update
if SIM == 1
    % Constant Bias
    bg = X(5:7,k);
else
    % Random Walk Bias
    bg = X(5:7,k) + dt*X(11:13,k);
end

% Rebuild the State Vector for Each Iteration
Xx(:,k) = [Q_kp1_n; bg];

% Calculated the Weighted Mean of the State Vector
x_bar = x_bar + wm(k)*Xx(:,k);
end

% Calculate Error Covariance
Px= zeros(nx, nx);
for k = 1:n
    Px = Px + wc(k)*((Xx(:,k) - x_bar)*(Xx(:,k) - x_bar)');
end

% Non-Linear Measurement Process Equations
Y = zeros(nR, n);                                % Pre-fill
y_bar = zeros(nR,1);                             % Pre-fill
Heading = zeros(1,n);                            % Pre-fill
if isempty(H1)
    H1 = zeros(1,n);                             % Pre-fill
    H2 = zeros(1,n);                             % Pre-fill
    Heading0 = zeros(1,n);                       % Pre-fill
    ii = zeros(1,n);                             % Pre-fill
end
for k = 1:n

if SIM == 1
    noise_R = X(11:13,k);
else
    noise_R = X(14:16,k);
end
```

```
% Quaternion to Euler Angles
q0 = Xx(1,k); q1 = Xx(2,k); q2 = Xx(3,k); q3 = Xx(4,k);
phi = atan2(2*(q2*q3+q0*q1), q0^2-q1^2-q2^2+q3^2) + noise_R(1);
theta = asin(-2*(q1*q3-q0*q2))+ noise_R(2);
psi = mod(atan2(2*(q1*q2+q0*q3), q0^2+q1^2-q2^2-q3^2)+ noise_R(3),
2*pi);

% Unwrap Heading Angle From [0, 2*pi] Range to Avoid Jumps
if nn == 1
    H1(k) = psi;
    Heading0(k) = H1(k);
    Heading(k) = H1(k);
else
    H2(k) = psi;
    if (H2(k)-H1(k)) <= -100*pi/180
        ii(k) = ii(k)+1;
    end
    if (H2(k)-H1(k)) >= 100*pi/180
        ii(k) = ii(k)-1;
    end
    Heading(k) = Heading0(k)+((H2(k)+2*pi*ii(k)) - Heading0(k));
    H1(k) = H2(k);
end

% Calculated Observations
Y(:,k)=[phi; theta; Heading(k)];

% Calculated Observation Mean
y_bar = y_bar + wm(k)*Y(:,k);
end

% Estimated Observation Covariance
Py = zeros(nR,nR);
for k = 1:n
    Py = Py + wc(k)*((Y(:,k) - y_bar)*(Y(:,k) - y_bar)');
end

% Estimated Cross Covariance
Pxy = zeros(nx,nR);
for k = 1:n
    Pxy = Pxy + wc(k)*((Xx(:,k) - x_bar)*(Y(:,k) - y_bar)');
end

% Measurement Method 1 (Accelerometer-GPS-Compass Fusion)
% Rotate GPS XYZ Acceleration about Heading
rx = ax_gps*cos(heading)+ay_gps*sin(heading);
ry = -ax_gps*sin(heading)+ay_gps*cos(heading);
rz = az_gps-g;
% Computer Pitch
theta = atan((-rx*rz - fx*sqrt(rx^2+rz^2-fx^2)) / (fx^2-rz^2));
% Compute Roll
r_theta = rx*sin(theta) + rz*cos(theta);
fc = fy-(norm(x_bar2(4:6)))*r; % Added Coriolis Term
```

```
phi = atan((r_theta*ry + fc*sqrt(ry^2+r_theta^2-fc^2)) / (fc^2-
r_theta^2));

% Heading Unwrap from [0, 2*pi]
if nn == 1
    H3 = heading;
    H4 = 0;
    psi0 = H3;
    psi_m = H3;
    jj=0;
else
    H4 = heading;
    if (H4-H3) <= -100*pi/180
        jj = jj+1;
    end
    if (H4-H3) >= 100*pi/180
        jj = jj-1;
    end
    psi_m = psi0+((H4+2*pi*jj) - psi0);
    H3 = H4;
end

% Measurement Vector
Z = [phi; theta; psi_m];

% Kalman Filter Equations
K = Pxy/Py;                                    % Kalman Gain
x_bar = x_bar + K*(Z - y_bar);                 % Kalman Correction

% Covariance Corrections
Px = Px - K*Py*K';

nn = 2;
%% INS ESTIMATOR

% Build Augmented State Vector
X = zeros(nxa2, n2);
xa = [x_bar2; zeros(nxa2-nx2,1)];

% Build Augmented Covariance Matrix
Pxa2 = [Px2 zeros(nx2,nQ2) zeros(nx2,nR2);
        zeros(nQ2,nx2) Qv2 zeros(nQ2,nR2);
        zeros(nR2,nx2) zeros(nR2,nQ2) R2];

% Construct 2n+1 SIGMA Vectors
c = eta2*Pxa2;
for k = 1:nxa2+1
    if k == 1
        X(:,k) = xa;
    else
        X(:,k) = xa + c(k-1,:)';
        X(:,k+nxa2) = xa - c(k-1,:)';
    end
end
```

```
% Time Update
Xx = zeros(nx2, n2);                                    % Pre-fill
x_bar2 = zeros(nx2, 1);                                 % Pre-fill
for k = 1:n2
% Rotation Matrix
R_t2b = rot_t2b(x_bar(1:4));
R_b2t = R_t2b';

% Position Update
P_kp1 = X(1:3,k) + dt*[X(4:5,k); -X(6,k)];

% Velocity and Bias Update
if SIM == 1
% No Sidereal
    V_kp1 = X(4:6,k) + dt*(R_b2t*([fx; fy; fz] - X(7:9,k) - ...
            X(10:12, k)) + [0; 0; g]);
    ba_kp1 = X(7:9,k);
else
% Sidereal Included
    V_kp1 = X(4:6,k) + dt*(R_b2t*([fx; fy; fz] - X(7:9,k) - ...
            X(10:12, k)) + [0; 0; g] - 2*cross([wei*cos(lat0); ...
            0; -wei*sin(lat0)], X(4:6,k)));
    ba_kp1 = X(7:9,k) + dt*X(13:15,k);
end

% States
Xx(:,k) = [P_kp1; V_kp1; ba_kp1];

% Calculated Mean
x_bar2 = x_bar2 + wm2(k)*Xx(:,k);
end

% Predicted Error Covariance
Px2= zeros(nx2, nx2);
for k = 1:n2
    Px2 = Px2 + wc2(k)*((Xx(:,k) - x_bar2)*(Xx(:,k) - x_bar2)');
end

% Measurement Equations
Y = zeros(nR2, n2);                                     % Pre-fill
y_bar = zeros(nR2,1);                                   % Pre-fill
for k = 1:n2

if SIM == 1
    % Calculated Observations
    Y(:,k)=Xx(1:6,k)+X(13:18,k);
else
    % Calculated Observations
    Y(:,k)=Xx(1:6,k)+X(16:21,k);
end

% Calculated Observation Mean
y_bar = y_bar + wm2(k)*Y(:,k);
```

```
end

% Estimated Observation Covariance
Py = zeros(nR2,nR2);
for k = 1:n2
    Py = Py + wc2(k)*((Y(:,k) - y_bar)*(Y(:,k) - y_bar)');
end

% Estimated Cross Covariance
Pxy = zeros(nx2,nR2);
for k = 1:n2
    Pxy = Pxy + wc2(k)*((Xx(:,k) - x_bar2)*(Y(:,k) - y_bar)');
end

% Measurements
Z = [N_sat; E_sat; D_sat; speed*cos(CoG); speed*sin(CoG); Vd_sat];

% Kalman Filter Equations
K = Pxy/Py;                                          % Kalman Gain
% Kalman State Corrections
x_bar2 = x_bar2 + K*(Z - y_bar);
% Covariance Correction
Px2 = Px2 - K*Py*K';

X_HAT = [x_bar(1:4); x_bar2(1:6); x_bar(5:7); x_bar2(7:9)];
end
kk=kk+1;
end

function R_t2b = rot_t2b(x)
q0 = x(1); q1 = x(2); q2 = x(3); q3 = x(4);
R_t2b = [q0^2+q1^2-q2^2-q3^2 2*(q1*q2+q0*q3) 2*(q1*q3-q0*q2);
         2*(q2*q1-q0*q3) q0^2-q1^2+q2^2-q3^2 2*(q2*q3+q0*q1);
         2*(q3*q1+q0*q2) 2*(q3*q2-q0*q1) q0^2-q1^2-q2^2+q3^2];
end

function q = e2q(phi, theta, psi)
% Form Rotation Matrix
R_psi = [cos(psi) sin(psi) 0; -sin(psi) cos(psi) 0; 0 0 1];
R_theta = [cos(theta) 0 -sin(theta); 0 1 0; sin(theta) 0 cos(theta)];
R_phi = [1 0 0; 0 cos(phi) sin(phi); 0 -sin(phi) cos(phi)];
R_n2b = R_phi*R_theta*R_psi;

% Extract Quaternions
q0=sqrt(1+trace(R_n2b))/2;
q1=(R_n2b(2,3)-R_n2b(3,2))/(4*q0);
q2=(R_n2b(3,1)-R_n2b(1,3))/(4*q0);
q3=(R_n2b(1,2)-R_n2b(2,1))/(4*q0);

q = [q0; q1; q2; q3];
end
```

2. Measurement Model Two Implementation

```
function X_HAT = UKF2(u)
%% UNSCENTED KALMAN FILTER
% Measurement Model 2
% LT Steven Terjesen
% September 2014

% This estimator is built for use in MATLAB Simulink. There are 39
inputs
% required and can be run in Simulink with an “Interpolated MATLAB
% Function” block. The first 13 inputs are the vehicle data: Course
Over
% Ground, Speed Over Ground, Accelerometer Measurements, Gyro
Measurements,
% GPS Measurements (In LTP XNorth-YEast-ZDown), Heading Measurement,
and
% Vertical Velocity (zeroed out for SEAFOXII data). Inputs 14 through
35
% are the Process Noise and Measurement Noise diagonal elements. These
% elements are not hard coded to allow for easier tuning. Inputs 36
% through 38 are the N-E-D accelerations in the LTP frame. Input 39 is
a
% toggle for selecting whether the data is from Condor or SEAFOX II.

%% System Inputs
% Measurement inputs
CoG = u(1);                                          %GPS Course Over Ground
speed = u(2);                                        %GPS Speed Over Ground
fx = u(3); fy = u(4); fz = u(5);                     %IMU Accelerations
p = u(6); q = u(7); r = u(8);                        %IMU Angular Rates
N_sat = u(9); E_sat = u(10); D_sat = u(11);          %GPS Position (NED)
heading = u(12);                                     %GPS Heading
Vd_sat = u(13);                                      %GPS LTP Down Velocity

% Process and Measurement Noise Matrices
R = diag(u(14:17));                                  %AHRS Measurement Noise
R2 = diag(u(18:23));                                 %INS Measurement Noise

SIM = u(39);                                         %Condor(1) or
SEAFOXII(2)
%                                                     Simulation Selector

if SIM == 1
    Qv = diag(u(24:26));                             %AHRS Process Noise
Condor
    Qv2 = diag(u(30:32));                            %INS Process Noise
Condor
else
    Qv = diag(u(24:29));                             %AHRS Process Noise
SEAFOX
    Qv2 = diag(u(30:35));                            %INS Process Noise
SEAFOX
    Vd_sat = 0;                                      %No Vertical Velocity
```

```
%                                                   measurement available
on
%                                                   SEAFOX II, set Vd=0.
    D_sat = 0;                                     %No Altitude
Measurement
%                                                   available on SEAFOXII,
set
%                                                   D_sat = 0;
end

% GPS Accelerations
ax_gps = u(36); ay_gps = u(37); az_gps = u(38); %GPS Accelerations as
                                                % calculated from 3rd
                                                % Order Filter


%% Initialization
persistent x_bar Heading0 H1 H2 ii jj psi0 H3 H4 ...
Px Px2 kk dt g nn nx nxa nQ nR n wc wm x_bar2 nx2 nxa2 n2 nQ2 nR2 wc2
...
wm2 eta eta2 wei lat0

% Allows time for Condor To steady out during live testing
if isempty(kk)
    kk = 0;
end
if kk < 1 && SIM==1
    X_HAT = zeros(16,1);
else

if isempty(x_bar)
% Miscellaneous
dt = 0.01;                                         %Filter dt [sec]
g = 9.8;                                           %Gravity Constant
[m/s^2]
wei = 7.292115*10^-5;                              %Sidereal Rate [rad/s]
lat0= 0.639268394832413;                           %Origin of LTP [rad]
%lon0 = -2.115435878466264
nn=1;                                              %Heading Count
Initializer

% AHRS Initialization
% Euler Angle Initialization
rx = ax_gps*cos(heading)+ay_gps*sin(heading);
ry = -ax_gps*sin(heading)+ay_gps*cos(heading);
rz = az_gps-g;
theta_0 = atan((-rx*rz - fx*sqrt(rx^2+rz^2-fx^2))/(fx^2-rz^2));
r_theta = rx*sin(theta_0) + rz*cos(theta_0);
fc = fy-speed*r;
phi_0 = atan((r_theta*ry + fc*sqrt(ry^2+r_theta^2-fc^2))/(fc^2-
r_theta^2));
psi_0=heading;
% Euler Angles to Quaternions
q_0 = e2q(phi_0, theta_0, psi_0);
% Gyro Bias Initial Values
```

```
bg_0 = [0; 0; 0];
% Inital State Vector
x_bar = [q_0; bg_0];
% Initial Covariance Estimate
Px = diag([1e-5*ones(1,4) 1e-3*ones(1,3)]);

%INS Initialization
% Position & Velocity Initialization
x_0 = N_sat;
y_0 = E_sat;
z_0 = D_sat;
vn_0 = speed*cos(CoG);
ve_0 = speed*sin(CoG);
vd_0 = Vd_sat;
% Accelerometer Initial Bias Estimate
ba_0 = [0; 0; 0];
% Initial State Vector
x_bar2 = [x_0; y_0; z_0; vn_0; ve_0; vd_0; ba_0];
% Initial Covariance Estimate
Px2 = diag([1e-5*ones(1,3), 1e-5*ones(1,3), 1e-3*ones(1,3)]);

% AHRS SIGMA Weights Initialization
nx = length(x_bar);                          % Number of AHRS States
nQ = length(diag(Qv));                       % Process Noise States
nR = length(diag(R));                        % Measurement Noise
States
nxa = nx+nQ+nR;                              % Total Augmented
States
n = 2*nxa+1;                                   % Number of
Iterations
alpha = 1e-3;                                % Tunable Scaling
Factor
beta = 2;                                    % beta = 2 for Gaussian
PDF
kappa = 0;                                   % Tunable Secondary
Scaling
%                                              Factor.
lamda = alpha^2*(nxa+kappa)-nxa;             % Weighting Factor
wm = (1/(2*(nxa+lamda)))*ones(n,1);          % Measurement Weights
wc = wm;                                     % Covariance Weights
wm(1) = lamda/(nxa+lamda);                   % Zeroth Measurement
Weight
wc(1) = wm(1) + (1-alpha^2+beta);            % Zeroth Covariance
Weight
eta = sqrt(nxa+lamda);                       % Covariance Weighting
%                                              Factor

% INS SIGMA Weights Initialization
nx2 = length(x_bar2);                        % Number of INS States
nQ2 = length(diag(Qv2));                     % Process Noise States
nR2 = length(diag(R2));                      % Measurement Noise
States
nxa2 = nx2+nQ2+nR2;                          % Total Augmented
States
n2 = 2*nxa2+1;                                 % Total Iterations
```

```
alpha2 = 1e-3;                                  % Tunable, 0<alpha<1
beta2 = 2;                                      % beta = 2 for Guassian
PDF
kappa2 = 0;                                     % Tunable Secondary
Scaling
%                                                 Factor.
lamda2 = alpha2^2*(nxa2+kappa2)-nxa2;           % Weighting Factor
wm2 = (1/(2*(nxa2+lamda2)))*ones(n2,1);         % Measurement Weights
wc2 = wm2;                                      % Covariance Weights
wm2(1) = lamda2/(nxa2+lamda2);                  % Zeroth Measurement
Weight
wc2(1) = wm2(1) + (1-alpha2^2+beta2);           % Zeroth Covariance
Weight
eta2 = sqrt(nxa2+lamda2);                       % Covariance Weighting
%                                                 Factor

end
%% AHRS ESTIMATOR

% Build Augmented State Vector
X = zeros(nxa, n);
xa = [x_bar; zeros(nxa-nx,1)];

% Build Augmented Covariance Matrix
Pxa = [Px zeros(nx,nQ) zeros(nx,nR);
       zeros(nQ,nx) Qv zeros(nQ,nR);
       zeros(nR,nx) zeros(nR,nQ) R];

% Build 2n+1 Sigma Vectors
c = eta*Pxa;
for k = 1:nxa+1
    if k == 1
        X(:,k) = xa;
    else
        X(:,k) = xa + c(k-1,:)';
        X(:,k+nxa) = xa - c(k-1,:)';
    end
end

% Time Update
Xx = zeros(nx, n);
x_bar = zeros(nx, 1);
for k = 1:n

% Gyro Model  w_bi(true) = w_bi(measured) - bias - noise
w_bi_b = [p-X(5,k)-X(8,k); q-X(6,k)-X(9,k); r-X(7, k)-X(10,k)];

if SIM == 1
    % No Sidereal Rate
    w_bt = w_bi_b;
else
    % Sidereal Rate Included
    R_t2b = rot_t2b(X(1:4,k));
    w_bt = w_bi_b - R_t2b*[wei*cos(lat0); 0; -wei*sin(lat0)];
```

```
end

% Quaternion Update
w1 = w_bt(1); w2 = w_bt(2); w3 = w_bt(3);
Q_kp1 = X(1:4, k) + (dt/2)*[0,  -w1,  -w2,  -w3;
                            w1,  0,    w3,  -w2;
                            w2, -w3,   0,    w1;
                            w3,  w2,  -w1,   0]*X(1:4, k);

% Quaternion Normalization
Q_kp1_n = Q_kp1/sqrt(Q_kp1'*Q_kp1);

% Gyro Bias Update
if SIM == 1
    % Constant Bias
    bg = X(5:7,k);
else
    % Random Walk Bias
    bg = X(5:7,k) + dt*X(11:13,k);
end

% Rebuild the State Vector for Each Iteration
Xx(:,k) = [Q_kp1_n; bg];

% Calculated the Weighted Mean of the State Vector
x_bar = x_bar + wm(k)*Xx(:,k);
end

% Calculate Error Covariance
Px= zeros(nx, nx);
for k = 1:n
    Px = Px + wc(k)*((Xx(:,k) - x_bar)*(Xx(:,k) - x_bar)');
end

% Non-Linear Measurement Process Equations
Y = zeros(nR, n);                                % Pre-fill
y_bar = zeros(nR,1);                             % Pre-fill
Heading = zeros(1,n);                            % Pre-fill
if isempty(H1)
    H1 = zeros(1,n);                             % Pre-fill
    H2 = zeros(1,n);                             % Pre-fill
    Heading0 = zeros(1,n);                       % Pre-fill
    ii = zeros(1,n);                             % Pre-fill
end
for k = 1:n

% Rotation Matrix (LTP to Body)
R_t2b = rot_t2b(Xx(1:4,k));
% Gyro Measurements
w_bi = [p; q; r] - Xx(5:7,k);
if SIM == 1
    noise_R = X(11:14,k);
    w_bt = w_bi;
else
```

```
    noise_R = X(14:17,k);
    w_bt = w_bi - R_t2b*[wei*cos(lat0); 0; -wei*sin(lat0)];
end

% Quaternion to Euler Angles
q0 = Xx(1,k); q1 = Xx(2,k); q2 = Xx(3,k); q3 = Xx(4,k);
psi = mod(atan2(2*(q1*q2+q0*q3), q0^2+q1^2-q2^2-q3^2)+ noise_R(4),
2*pi);
% Unwrap Heading Angle From [0, 2*pi] Range to Aviod Jumps
if nn == 1
    H1(k) = psi;
    Heading0(k) = H1(k);
    Heading(k) = H1(k);
else
    H2(k) = psi;
    if (H2(k)-H1(k)) <= -100*pi/180
        ii(k) = ii(k)+1;
    end
    if (H2(k)-H1(k)) >= 100*pi/180
        ii(k) = ii(k)-1;
    end
    Heading(k) = Heading0(k)+((H2(k)+2*pi*ii(k)) - Heading0(k));
    H1(k) = H2(k);
end

% Accelerometer Estimate
FB_hat = R_t2b*[ax_gps; ay_gps; az_gps] + ...
        cross(w_bt, R_t2b*x_bar2(4:6)) - ...
        R_t2b*[0; 0; g]-x_bar2(7:9)-noise_R(1:3);

% Calculated Observations
Y(:,k)=[FB_hat; Heading(k)];

% Calculated Observation Mean
y_bar = y_bar + wm(k)*Y(:,k);
end

% Estimated Observation Covariance
Py = zeros(nR,nR);
for k = 1:n
    Py = Py + wc(k)*((Y(:,k) - y_bar)*(Y(:,k) - y_bar)');
end

% Estimated Cross Covariance
Pxy = zeros(nx,nR);
for k = 1:n
    Pxy = Pxy + wc(k)*((Xx(:,k) - x_bar)*(Y(:,k) - y_bar)');
end

% Measurement Method 2 (Accelerometer)
% Heading Unwrap from [0, 2*pi]
if nn == 1
    H3 = heading;
    H4 = 0;
```

```
    psi0 = H3;
    psi_m = H3;
    jj=0;
else
    H4 = heading;
    if (H4-H3) <= -100*pi/180
        jj = jj+1;
    end
    if (H4-H3) >= 100*pi/180
        jj = jj-1;
    end
    psi_m = psi0+((H4+2*pi*jj) - psi0);
    H3 = H4;
end

% Measurement Vector
Z = [fx; fy; fz; psi_m];

% Kalman Filter Equations
K = Pxy/Py;                                    % Kalman Gain
x_bar = x_bar + K*(Z - y_bar);                 % Kalman Correction

% Covariance Corrections
Px = Px - K*Py*K';

nn = 2;
%% INS ESTIMATOR

% Build Augmented State Vector
X = zeros(nxa2, n2);
xa = [x_bar2; zeros(nxa2-nx2,1)];

% Build Augmented Covariance Matrix
Pxa2 = [Px2 zeros(nx2,nQ2) zeros(nx2,nR2);
        zeros(nQ2,nx2) Qv2 zeros(nQ2,nR2);
        zeros(nR2,nx2) zeros(nR2,nQ2) R2];

% Construct 2n+1 SIGMA Vectors
c = eta2*Pxa2;
for k = 1:nxa2+1
    if k == 1
        X(:,k) = xa;
    else
        X(:,k) = xa + c(k-1,:)';
        X(:,k+nxa2) = xa - c(k-1,:)';
    end
end

% Time Update
Xx = zeros(nx2, n2);                           % Pre-fill
x_bar2 = zeros(nx2, 1);                        % Pre-fill
for k = 1:n2
% Rotation Matrix
R_t2b = rot_t2b(x_bar(1:4));
```

```
R_b2t = R_t2b';

% Posittion Update
P_kp1 = X(1:3,k) + dt*[X(4:5,k); -X(6,k)];

% Velocity and Bias Update
if SIM == 1
% No Sidereal
    V_kp1 = X(4:6,k) + dt*(R_b2t*([fx; fy; fz] - X(7:9,k) - ...
            X(10:12, k)) + [0; 0; g]);
    ba_kp1 = X(7:9,k);
else
% Sidereal Included
    V_kp1 = X(4:6,k) + dt*(R_b2t*([fx; fy; fz] - X(7:9,k) - ...
            X(10:12, k)) + [0; 0; g] - 2*cross([wei*cos(lat0); ...
            0; -wei*sin(lat0)], X(4:6,k)));
    ba_kp1 = X(7:9,k) + dt*X(13:15,k);
end

% States
Xx(:,k) = [P_kp1; V_kp1; ba_kp1];

% Calculated Mean
x_bar2 = x_bar2 + wm2(k)*Xx(:,k);
end

% Predicted Error Covariance
Px2= zeros(nx2, nx2);
for k = 1:n2
    Px2 = Px2 + wc2(k)*((Xx(:,k) - x_bar2)*(Xx(:,k) - x_bar2)');
end

% Measurement Equations
Y = zeros(nR2, n2);                                 % Pre-fill
y_bar = zeros(nR2,1);                               % Pre-fill
for k = 1:n2

if SIM == 1
    % Calculated Observations
    Y(:,k)=Xx(1:6,k)+X(13:18,k);
else
    % Calculated Observations
    Y(:,k)=Xx(1:6,k)+X(16:21,k);
end

% Calculated Observation Mean
y_bar = y_bar + wm2(k)*Y(:,k);
end

% Estimated Observation Covariance
Py = zeros(nR2,nR2);
for k = 1:n2
    Py = Py + wc2(k)*((Y(:,k) - y_bar)*(Y(:,k) - y_bar)');
end
```

```
% Estimated Cross Covariance
Pxy = zeros(nx2,nR2);
for k = 1:n2
    Pxy = Pxy + wc2(k)*((Xx(:,k) - x_bar2)*(Y(:,k) - y_bar)');
end

% Measurements
Z = [N_sat; E_sat; D_sat; speed*cos(CoG); speed*sin(CoG); Vd_sat];

% Kalman Filter Equations
K = Pxy/Py;                                          % Kalman Gain
% Kalman State Corrections
x_bar2 = x_bar2 + K*(Z - y_bar);
% Covariance Correction
Px2 = Px2 - K*Py*K';

X_HAT = [x_bar(1:4); x_bar2(1:6); x_bar(5:7); x_bar2(7:9)];
end
kk=kk+1;
end

function R_t2b = rot_t2b(x)
q0 = x(1); q1 = x(2); q2 = x(3); q3 = x(4);
R_t2b = [q0^2+q1^2-q2^2-q3^2 2*(q1*q2+q0*q3) 2*(q1*q3-q0*q2);
         2*(q2*q1-q0*q3) q0^2-q1^2+q2^2-q3^2 2*(q2*q3+q0*q1);
         2*(q3*q1+q0*q2) 2*(q3*q2-q0*q1) q0^2-q1^2-q2^2+q3^2];
end

function q = e2q(phi, theta, psi)
% Form Rotation Matrix
R_psi = [cos(psi) sin(psi) 0; -sin(psi) cos(psi) 0; 0 0 1];
R_theta = [cos(theta) 0 -sin(theta); 0 1 0; sin(theta) 0 cos(theta)];
R_phi = [1 0 0; 0 cos(phi) sin(phi); 0 -sin(phi) cos(phi)];
R_n2b = R_phi*R_theta*R_psi;

% Extract Quaternions
q0=sqrt(1+trace(R_n2b))/2;
q1=(R_n2b(2,3)-R_n2b(3,2))/(4*q0);
q2=(R_n2b(3,1)-R_n2b(1,3))/(4*q0);
q3=(R_n2b(1,2)-R_n2b(2,1))/(4*q0);

q = [q0; q1; q2; q3];
end
```

D. SQUARE ROOT UNSCENTED KALMAN FILTER MATLAB CODE

1. Measurement Model One Implementation

```
function X_HAT = SRUKF1(u)
%% SQUARE ROOOT UNSCENTED KALMAN FILTER
% Measurement Model 1
% LT Steven Terjesen
```

```
% September 2014

% This estimator is built for use in MATLAB Simulink. There are 38
inputs
% required and can be run in Simulink with an “Interpolated MATLAB
% Function” block. The first 13 inputs are the vehicle data: Course
Over
% Ground, Speed Over Ground, Accelerometer Measurements, Gyro
Measurements,
% GPS Measurements (In LTP XNorth-YEast-ZDown), Heading Measurement,
and
% Vertical Velocity (zeroed out for SEAFOXII data). Inputs 14 through
34
% are the Process Noise and Measurement Noise diagonal elements. These
% elements are not hard coded to allow for easier tuning. Inputs 35
% through 37 are the N-E-D accelerations in the LTP frame. Input 38 is
a
% toggle for selecting whether the data is from Condor or SEAFOX II.

%% System Inputs
% Measurement inputs
CoG = u(1);                                         %GPS Course Over Ground
speed = u(2);                                       %GPS Speed Over Ground
fx = u(3); fy = u(4); fz = u(5);                    %IMU Accelerations
p = u(6); q = u(7); r = u(8);                       %IMU Angular Rates
N_sat = u(9); E_sat = u(10); D_sat = u(11);         %GPS Position (NED)
heading = u(12);                                    %GPS Heading
Vd_sat = u(13);                                     %GPS LTP Down Velocity

% Process and Measurement Noise Matrices
R = diag(u(14:16));                                 %AHRS Measurement Noise
R2 = diag(u(17:22));                                %INS Measurement Noise

SIM = u(38);                                        %Condor(1) or
SEAFOXII(2)
%                                                    Simulation Selector

if SIM == 1
    Qv = diag(u(23:25));                            %AHRS Process Noise
Condor
    Qv2 = diag(u(29:31));                           %INS Process Noise
Condor
else
    Qv = diag(u(23:28));                            %AHRS Process Noise
SEAFOX
    Qv2 = diag(u(29:34));                           %INS Process Noise
SEAFOX
    Vd_sat = 0;                                     %No Vertical Velocity
%                                                    measurement available
on
%                                                    SEAFOX II, set Vd=0.
    D_sat = 0;                                      %No Altitude
Measurement
```

```
%                                                   available on SEAFOXII,
set
%                                                   D_sat = 0;
end

% GPS Accelerations
ax_gps = u(35); ay_gps = u(36); az_gps = u(37); %GPS Accelerations as
                                                % calculated from 3rd
                                                % Order Filter

%% Initialization
persistent x_bar Heading0 H1 H2 ii jj psi0 H3 H4 ...
Sx Sx2 kk dt g nn nx nxa nQ nR n wc wm x_bar2 nx2 nxa2 n2 nQ2 nR2 wc2
...
wm2 wei lat0 eta eta2

% Allows time for Condor To steady out during live testing
if isempty(kk)
    kk = 0;
end
if kk < 1 && SIM==1
    X_HAT = zeros(16,1);
else

if isempty(x_bar)
% Miscellaneous
dt = 0.01;                                          %Filter dt [sec]
g = 9.8;                                            %Gravity Constant
[m/s^2]
wei = 7.292115*10^-5;                               %Sidereal Rate [rad/s]
lat0= 0.639268394832413;                            %Origin of LTP [rad]
%lon0 = -2.115435878466264
nn=1;                                               %Heading Count
Initializer

% AHRS Initialization
% Euler Angle Initialization
rx = ax_gps*cos(heading)+ay_gps*sin(heading);
ry = -ax_gps*sin(heading)+ay_gps*cos(heading);
rz = az_gps-g;
theta_0 = atan((-rx*rz - fx*sqrt(rx^2+rz^2-fx^2))/(fx^2-rz^2));
r_theta = rx*sin(theta_0) + rz*cos(theta_0);
fc = fy-speed*r;
phi_0 = atan((r_theta*ry + fc*sqrt(ry^2+r_theta^2-fc^2))/(fc^2-
r_theta^2));
psi_0=heading;
% Euler Angles to Quaternions
q_0 = e2q(phi_0, theta_0, psi_0);
% Gyro Bias Initial Values
bg_0 = [0; 0; 0];
% Inital State Vector
x_bar = [q_0; bg_0];
% Initial Covariance Estimate
Px = diag([1e-5*ones(1,4) 1e-3*ones(1,3)]);
```

```
% Inital Matrix Square Root
Sx = chol(Px);

%INS Initialization
% Position & Velocity Initialization
x_0 = N_sat;
y_0 = E_sat;
z_0 = D_sat;
vn_0 = speed*cos(CoG);
ve_0 = speed*sin(CoG);
vd_0 = Vd_sat;
% Accelerometer Initial Bias Estimate
ba_0 = [0; 0; 0];
% Initial State Vector
x_bar2 = [x_0; y_0; z_0; vn_0; ve_0; vd_0; ba_0];
% Initial Covariance Estimate
Px2 = diag([1e-5*ones(1,3), 1e-5*ones(1,3), 1e-3*ones(1,3)]);
% Inital Matrix Square Root
Sx2 = chol(Px2);

% AHRS SIGMA Weights Initialization
nx = length(x_bar);                          % Number of AHRS States
nQ = length(diag(Qv));                       % Process Noise States
nR = length(diag(R));                        % Measurement Noise
States
nxa = nx+nQ+nR;                              % Total Augmented
States
n = 2*nxa+1;                                   % Number of
Iterations
alpha = 1e-3;                                % Tunable Scaling
Factor
beta = 2;                                    % beta = 2 for Gaussian
PDF
kappa = 0;                                   % Tunable Secondary
Scaling
%                                              Factor.
lamda = alpha^2*(nxa+kappa)-nxa;             % Weighting Factor
wm = (1/(2*(nxa+lamda)))*ones(n,1);          % Measurement Weights
wc = wm;                                     % Covariance Weights
wm(1) = lamda/(nxa+lamda);                   % Zeroth Measurement
Weight
wc(1) = wm(1) + (1-alpha^2+beta);            % Zeroth Covariance
Weight
eta = sqrt(nxa+lamda);                       % Covariance Weighting
%                                              Factor

% INS SIGMA Weights Initialization
nx2 = length(x_bar2);                        % Number of INS States
nQ2 = length(diag(Qv2));                     % Process Noise States
nR2 = length(diag(R2));                      % Measurement Noise
States
nxa2 = nx2+nQ2+nR2;                          % Total Augmented
States
n2 = 2*nxa2+1;                                 % Total Iterations
alpha2 = 1e-3;                               % Tunable, 0<alpha<1
```

```
beta2 = 2;                                          % beta = 2 for Gaussian
PDF
kappa2 = 0;                                         % Tunable Secondary
Scaling
%                                                     Factor.
lamda2 = alpha2^2*(nxa2+kappa2)-nxa2;               % Weighting Factor
wm2 = (1/(2*(nxa2+lamda2)))*ones(n2,1);             % Measurement Weights
wc2 = wm2;                                          % Covariance Weights
wm2(1) = lamda2/(nxa2+lamda2);                      % Zeroth Measurement
Weight
wc2(1) = wm2(1) + (1-alpha2^2+beta2);               % Zeroth Covariance
Weight
eta2 = sqrt(nxa2+lamda2);                           % Covariance Weighting
%                                                     Factor
end
%% AHRS ESTIMATOR

% Build Augmented State Vector
X = zeros(nxa, n);
xa = [x_bar; zeros(nxa-nx,1)];

% Build Augmented Covariance Matrix
Sxa = [Sx zeros(nx,nQ) zeros(nx,nR);
        zeros(nQ,nx) Qv zeros(nQ,nR);
        zeros(nR,nx) zeros(nR,nQ) R];

% Build 2n+1 Sigma Vectors
c = eta*Sxa;
for k = 1:nxa+1
    if k == 1
        X(:,k) = xa;
    else
        X(:,k) = xa + c(k-1,:)';
        X(:,k+nxa) = xa - c(k-1,:)';
    end
end

% Time Update
Xx = zeros(nx, n);
x_bar = zeros(nx, 1);
for k = 1:n

% Gyro Model  w_bi(true) = w_bi(measured) - bias - noise
w_bi_b = [p-X(5,k)-X(8,k); q-X(6,k)-X(9,k); r-X(7, k)-X(10,k)];

if SIM == 1
    % No Sidereal Rate
    w_bt = w_bi_b;
else
    % Sidereal Rate Included
    R_t2b = rot_t2b(X(1:4,k));
    w_bt = w_bi_b - R_t2b*[wei*cos(lat0); 0; -wei*sin(lat0)];
end
```

```
% Quaternion Update
w1 = w_bt(1); w2 = w_bt(2); w3 = w_bt(3);
Q_kp1 = X(1:4, k) + (dt/2)*[0,  -w1,  -w2,  -w3;
                           w1,  0,    w3,  -w2;
                           w2, -w3,   0,    w1;
                           w3,  w2,  -w1,   0]*X(1:4, k);

% Quaternion Normalization
Q_kp1_n = Q_kp1/sqrt(Q_kp1'*Q_kp1);

% Gyro Bias Update
if SIM == 1
    % Constant Bias
    bg = X(5:7,k);
else
    % Random Walk Bias
    bg = X(5:7,k) + dt*X(11:13,k);
end

% Rebuild the State Vector for Each Iteration
Xx(:,k) = [Q_kp1_n; bg];

% Calculated the Weighted Mean of the State Vector
x_bar = x_bar + wm(k)*Xx(:,k);
end

% Calculate Error Covariance
Ax = sqrt(wc(2))*(Xx(:,2:n) - x_bar(:,ones(1,n-1)));
% QR Decomposition
[~, sx] = qr(Ax',0);
% Cholesky Update (Downdate for negative zeroth weight)
Sx = cholupdate(sx, sqrt(-wc(1))*(Xx(:,1)-x_bar), '-');

% Non-Linear Measurement Process Equations
Y = zeros(nR, n);                                % Pre-fill
y_bar = zeros(nR,1);                             % Pre-fill
Heading = zeros(1,n);                            % Pre-fill
if isempty(H1)
    H1 = zeros(1,n);                             % Pre-fill
    H2 = zeros(1,n);                             % Pre-fill
    Heading0 = zeros(1,n);                       % Pre-fill
    ii = zeros(1,n);                             % Pre-fill
end
for k = 1:n

if SIM == 1
    noise_R = X(11:13,k);
else
    noise_R = X(14:16,k);
end

% Quaternion to Euler Angles
q0 = Xx(1,k); q1 = Xx(2,k); q2 = Xx(3,k); q3 = Xx(4,k);
phi = atan2(2*(q2*q3+q0*q1), q0^2-q1^2-q2^2+q3^2) + noise_R(1);
```

```
theta = asin(-2*(q1*q3-q0*q2))+ noise_R(2);
psi = mod(atan2(2*(q1*q2+q0*q3), q0^2+q1^2-q2^2-q3^2)+ noise_R(3),
2*pi);

% Unwrap Heading Angle From [0, 2*pi] Range to Avoid Jumps
if nn == 1
    H1(k) = psi;
    Heading0(k) = H1(k);
    Heading(k) = H1(k);
else
    H2(k) = psi;
    if (H2(k)-H1(k)) <= -100*pi/180
        ii(k) = ii(k)+1;
    end
    if (H2(k)-H1(k)) >= 100*pi/180
        ii(k) = ii(k)-1;
    end
    Heading(k) = Heading0(k)+((H2(k)+2*pi*ii(k)) - Heading0(k));
    H1(k) = H2(k);
end

% Calculated Observations
Y(:,k)=[phi; theta; Heading(k)];

% Calculated Observation Mean
y_bar = y_bar + wm(k)*Y(:,k);
end

% Estimated Observation Covariance
Bx = sqrt(wc(2))*(Y(:,2:n) - y_bar(:,ones(1,n-1)));
% QR Decomposition
[~, sy] = qr(Bx',0); % **"qr" PRODUCES UPPER TRIANGULAR MATRIX**
% Cholesky Update (or Downdate)
Sy = cholupdate(sy, sqrt(-wc(1))*(Y(:,1)-y_bar), '-');
% ***Sy MUST BE LOWER TRIANGULAR***
Sy = Sy';

% Estimated Cross Covariance
Pxy = zeros(nx,nR);
for k = 1:n
    Pxy = Pxy + wc(k)*((Xx(:,k) - x_bar)*(Y(:,k) - y_bar)');
end

% Measurement Method 1 (Accelerometer-GPS-Compass Fusion)
% Rotate GPS XYZ Acceleration about Heading
rx = ax_gps*cos(heading)+ay_gps*sin(heading);
ry = -ax_gps*sin(heading)+ay_gps*cos(heading);
rz = az_gps-g;
% Computer Pitch
theta = atan((-rx*rz - fx*sqrt(rx^2+rz^2-fx^2)) / (fx^2-rz^2));
% Compute Roll
r_theta = rx*sin(theta) + rz*cos(theta);
fc = fy-(norm(x_bar2(4:6)))*r; % Added Coriolis Term
```

```
phi = atan((r_theta*ry + fc*sqrt(ry^2+r_theta^2-fc^2)) / (fc^2-
r_theta^2));

% Heading Unwrap from [0, 2*pi]
if nn == 1
    H3 = heading;
    H4 = 0;
    psi0 = H3;
    psi_m = H3;
    jj=0;
else
    H4 = heading;
    if (H4-H3) <= -100*pi/180
        jj = jj+1;
    end
    if (H4-H3) >= 100*pi/180
        jj = jj-1;
    end
    psi_m = psi0+((H4+2*pi*jj) - psi0);
    H3 = H4;
end

% Measurement Vector
Z = [phi; theta; psi_m];

% Kalman Filter Equations
K = (Pxy/Sy')/Sy;                              % Efficient Least
Squares
x_bar = x_bar + K*(Z - y_bar);                 % Kalman Correction

% Covariance Corrections
Ux = K*Sy;
for k = 1:nR
    Sx = cholupdate(Sx, Ux(:,k), '-');         % Cholesky Downdate
end
Sx = Sx';                                      % MUST BE LOWER
TRIANGULAR

nn = 2;
%% INS ESTIMATOR

% Build Augmented State Vector
X = zeros(nxa2, n2);
xa = [x_bar2; zeros(nxa2-nx2,1)];

% Build Augmented Covariance Matrix
Sxa2 = [Sx2 zeros(nx2,nQ2) zeros(nx2,nR2);
        zeros(nQ2,nx2) Qv2 zeros(nQ2,nR2);
        zeros(nR2,nx2) zeros(nR2,nQ2) R2];

% Construct n+2 SIGMA Vectors
c = eta2*Sxa2;
for k = 1:nxa2+1
    if k == 1
```

```
        X(:,k) = xa;
    else
        X(:,k) = xa + c(k-1,:)';
        X(:,k+nxa2) = xa - c(k-1,:)';
    end
end

% Time Update
Xx = zeros(nx2, n2);                                % Pre-fill
x_bar2 = zeros(nx2, 1);                             % Pre-fill
for k = 1:n2
% Rotation Matrix
R_t2b = rot_t2b(x_bar(1:4));
R_b2t = R_t2b';

% Position Update
P_kp1 = X(1:3,k) + dt*[X(4:5,k); -X(6,k)];

% Velocity and Bias Update
if SIM == 1
% No Sidereal
    V_kp1 = X(4:6,k) + dt*(R_b2t*([fx; fy; fz] - X(7:9,k) - ...
            X(10:12, k)) + [0; 0; g]);
    ba_kp1 = X(7:9,k);
else
% Sidereal Included
    V_kp1 = X(4:6,k) + dt*(R_b2t*([fx; fy; fz] - X(7:9,k) - ...
            X(10:12, k)) + [0; 0; g] - 2*cross([wei*cos(lat0); ...
            0; -wei*sin(lat0)], X(4:6,k)));
    ba_kp1 = X(7:9,k) + dt*X(13:15,k);
end

% States
Xx(:,k) = [P_kp1; V_kp1; ba_kp1];

% Calculated Mean
x_bar2 = x_bar2 + wm2(k)*Xx(:,k);
end

% Predicted Error Covariance
Ax = sqrt(wc2(2))*(Xx(:,2:n2) - x_bar2(:,ones(1,n2-1)));
% QR Decomposition
[~, Sx2] = qr(Ax',0);
% Cholesky Factor Update
Sx2 = cholupdate(Sx2, sqrt(-wc2(1))*(Xx(:,1)-x_bar2), '-');

% Measurement Equations
Y = zeros(nR2, n2);                                 % Pre-fill
y_bar = zeros(nR2,1);                               % Pre-fill
for k = 1:n2

if SIM == 1
    % Calculated Observations
    Y(:,k)=Xx(1:6,k)+X(13:18,k);
```

```
else
    % Calculated Observations
    Y(:,k)=Xx(1:6,k)+X(16:21,k);
end

% Calculated Observation Mean
y_bar = y_bar + wm2(k)*Y(:,k);
end

% Estimated Observation Covariance
Bx = sqrt(wc2(2))*(Y(:,2:n2) - y_bar(:,ones(1,n2-1)));
% QR Decomposition
[~, Sy] = qr(Bx',0);
% Cholesky Factor Update
Sy = cholupdate(Sy, sqrt(-wc2(1))*(Y(:,1)-y_bar), '-');
% Make Sy Lower Triangular
Sy = Sy';

% Estimated Cross Covariance
Pxy = zeros(nx2,nR2);
for k = 1:n2
    Pxy = Pxy + wc2(k)*((Xx(:,k) - x_bar2)*(Y(:,k) - y_bar)');
end

% Measurements
Z = [N_sat; E_sat; D_sat; speed*cos(CoG); speed*sin(CoG); Vd_sat];

% Kalman Filter Equations
% Kalman Gain w/ Efficient Least Squares
K = (Pxy/Sy')/Sy;
% Kalman State Corrections
x_bar2 = x_bar2 + K*(Z - y_bar);
% Covariance Correction
Ux = K*Sy;
for k = 1:nR2
    Sx2 = cholupdate(Sx2, Ux(:,k), '-');
end
% Make Sx2 Lower Right Triangular
Sx2 = Sx2';

X_HAT = [x_bar(1:4); x_bar2(1:6); x_bar(5:7); x_bar2(7:9)];
end
kk=kk+1;
end

function R_t2b = rot_t2b(x)
q0 = x(1); q1 = x(2); q2 = x(3); q3 = x(4);
R_t2b = [q0^2+q1^2-q2^2-q3^2 2*(q1*q2+q0*q3) 2*(q1*q3-q0*q2);
         2*(q2*q1-q0*q3) q0^2-q1^2+q2^2-q3^2 2*(q2*q3+q0*q1);
         2*(q3*q1+q0*q2) 2*(q3*q2-q0*q1) q0^2-q1^2-q2^2+q3^2];
end

function q = e2q(phi, theta, psi)
% Form Rotation Matrix
```

```
R_psi = [cos(psi) sin(psi) 0; -sin(psi) cos(psi) 0; 0 0 1];
R_theta = [cos(theta) 0 -sin(theta); 0 1 0; sin(theta) 0 cos(theta)];
R_phi = [1 0 0; 0 cos(phi) sin(phi); 0 -sin(phi) cos(phi)];
R_n2b = R_phi*R_theta*R_psi;

% Extract Quaternions
q0=sqrt(1+trace(R_n2b))/2;
q1=(R_n2b(2,3)-R_n2b(3,2))/(4*q0);
q2=(R_n2b(3,1)-R_n2b(1,3))/(4*q0);
q3=(R_n2b(1,2)-R_n2b(2,1))/(4*q0);

q = [q0; q1; q2; q3];
end
```

2. Measurement Model Two Implementation

```
function X_HAT = SRUKF2(u)
%% SQUARE ROOOT UNSCENTED KALMAN FILTER
% Measurement Model 2
% LT Steven Terjesen
% September 2014

% This estimator is built for use in MATLAB Simulink. There are 39
inputs
% required and can be run in Simulink with an “Interpolated MATLAB
% Function” block. The first 13 inputs are the vehicle data: Course
Over
% Ground, Speed Over Ground, Accelerometer Measurements, Gyro
Measurements,
% GPS Measurements (In LTP XNorth-YEast-ZDown), Heading Measurement,
and
% Vertical Velocity (zeroed out for SEAFOXII data). Inputs 14 through
35
% are the Process Noise and Measurement Noise diagonal elements. These
% elements are not hard coded to allow for easier tuning. Inputs 36
% through 38 are the N-E-D accelerations in the LTP frame. Input 39 is
a
% toggle for selecting whether the data is from Condor or SEAFOX II.

%% System Inputs
% Measurement inputs
CoG = u(1);                                          %GPS Course Over Ground
speed = u(2);                                        %GPS Speed Over Ground
fx = u(3); fy = u(4); fz = u(5);                     %IMU Accelerations
p = u(6); q = u(7); r = u(8);                        %IMU Angular Rates
N_sat = u(9); E_sat = u(10); D_sat = u(11);          %GPS Position (NED)
heading = u(12);                                     %GPS Heading
Vd_sat = u(13);                                      %GPS LTP Down Velocity

% Process and Measurement Noise Matrices
R = chol(diag(u(14:17)));                            %AHRS Measurement Noise
R2 = chol(diag(u(18:23)));                           %INS Measurement Noise
```

```
SIM = u(39);                                        %Condor(1) or
SEAFOXII(2)
%                                                    Simulation Selector

if SIM == 1
    Qv = chol(diag(u(24:26)));                      %AHRS Process Noise
Condor
    Qv2 = chol(diag(u(30:32)));                     %INS Process Noise
Condor
else
    Qv = chol(diag(u(24:29)));                      %AHRS Process Noise
SEAFOX
    Qv2 = chol(diag(u(30:35)));                     %INS Process Noise
SEAFOX
    Vd_sat = 0;                                     %No Vertical Velocity
%                                                    measurement available
on
%                                                    SEAFOX II, set Vd=0.
    D_sat = 0;                                      %No Altitude
Measurement
%                                                    available on SEAFOXII,
set
%                                                    D_sat = 0;
end

% GPS Accelerations
ax_gps = u(36); ay_gps = u(37); az_gps = u(38); %GPS Accelerations as
                                                % calculated from 3rd
                                                % Order Filter


%% Initialization
persistent x_bar Heading0 H1 H2 ii jj psi0 H3 H4 ...
Sx Sx2 kk dt g nn nx nxa nQ nR n wc wm x_bar2 nx2 nxa2 n2 nQ2 nR2 wc2
...
wm2 wei lat0 eta eta2

% Allows time for Condor To steady out during live testing
if isempty(kk)
    kk = 0;
end
if kk < 1 && SIM==1
    X_HAT = zeros(16,1);
else

if isempty(x_bar)
% Miscellaneous
dt = 0.01;                                          %Filter dt [sec]
g = 9.8;                                            %Gravity Constant
[m/s^2]
wei = 7.292115*10^-5;                               %Sidereal Rate [rad/s]
lat0= 0.639268394832413;                            %Origin of LTP [rad]
%lon0 = -2.115435878466264
nn=1;                                               %Heading Count
Initializer
```

```
% AHRS Initialization
% Euler Angle Initialization
rx = ax_gps*cos(heading)+ay_gps*sin(heading);
ry = -ax_gps*sin(heading)+ay_gps*cos(heading);
rz = az_gps-g;
theta_0 = atan((-rx*rz - fx*sqrt(rx^2+rz^2-fx^2))/(fx^2-rz^2));
r_theta = rx*sin(theta_0) + rz*cos(theta_0);
fc = fy-speed*r;
phi_0 = atan((r_theta*ry + fc*sqrt(ry^2+r_theta^2-fc^2))/(fc^2-
r_theta^2));
psi_0=heading;
% Euler Angles to Quaternions
q_0 = e2q(phi_0, theta_0, psi_0);
% Gyro Bias Initial Values
bg_0 = [0; 0; 0];
% Inital State Vector
x_bar = [q_0; bg_0];
% Initial Covariance Estimate
Px = diag([1e-5*ones(1,4) 1e-3*ones(1,3)]);
% Inital Matrix Square Root
Sx = chol(Px);

%INS Initialization
% Position & Velocity Initialization
x_0 = N_sat;
y_0 = E_sat;
z_0 = D_sat;
vn_0 = speed*cos(CoG);
ve_0 = speed*sin(CoG);
vd_0 = Vd_sat;
% Accelerometer Initial Bias Estimate
ba_0 = [0; 0; 0];
% Initial State Vector
x_bar2 = [x_0; y_0; z_0; vn_0; ve_0; vd_0; ba_0];
% Initial Covariance Estimate
Px2 = diag([1e-5*ones(1,3), 1e-5*ones(1,3), 1e-3*ones(1,3)]);
% Inital Matrix Square Root
Sx2 = chol(Px2);

% AHRS SIGMA Weights Initialization
nx = length(x_bar);                              % Number of AHRS States
nQ = length(diag(Qv));                           % Process Noise States
nR = length(diag(R));                            % Measurement Noise
States
nxa = nx+nQ+nR;                                  % Total Augmented
States
n = 2*nxa+1;                                       % Number of
Iterations
alpha = 1e-3;                                    % Tunable Scaling
Factor
beta = 2;                                        % beta = 2 for Gaussian
PDF
kappa = 0;                                       % Tunable Secondary
Scaling
```

```
%                                                   Factor.
lamda = alpha^2*(nxa+kappa)-nxa;                  % Weighting Factor
wm = (1/(2*(nxa+lamda)))*ones(n,1);               % Measurement Weights
wc = wm;                                          % Covariance Weights
wm(1) = lamda/(nxa+lamda);                        % Zeroth Measurement
Weight
wc(1) = wm(1) + (1-alpha^2+beta);                 % Zeroth Covariance
Weight
eta = sqrt(nxa+lamda);                            % Covariance Weighting
%                                                   Factor

% INS SIGMA Weights Initialization
nx2 = length(x_bar2);                             % Number of INS States
nQ2 = length(diag(Qv2));                          % Process Noise States
nR2 = length(diag(R2));                           % Measurement Noise
States
nxa2 = nx2+nQ2+nR2;                               % Total Augmented
States
n2 = 2*nxa2+1;                                      % Total Iterations
alpha2 = 1e-3;                                    % Tunable, 0<alpha<1
beta2 = 2;                                        % beta = 2 for Gaussian
PDF
kappa2 = 0;                                       % Tunable Secondary
Scaling
%                                                   Factor.
lamda2 = alpha2^2*(nxa2+kappa2)-nxa2;             % Weighting Factor
wm2 = (1/(2*(nxa2+lamda2)))*ones(n2,1);           % Measurement Weights
wc2 = wm2;                                        % Covariance Weights
wm2(1) = lamda2/(nxa2+lamda2);                    % Zeroth Measurement
Weight
wc2(1) = wm2(1) + (1-alpha2^2+beta2);             % Zeroth Covariance
Weight
eta2 = sqrt(nxa2+lamda2);                         % Covariance Weighting
%                                                   Factor

end
%% AHRS ESTIMATOR

% Build Augmented State Vector
X = zeros(nxa, n);
xa = [x_bar; zeros(nxa-nx,1)];

% Build Augmented Covariance Matrix
Sxa = [Sx zeros(nx,nQ) zeros(nx,nR);
       zeros(nQ,nx) Qv zeros(nQ,nR);
       zeros(nR,nx) zeros(nR,nQ) R];

% Build n+2 Sigma Vectors
c = eta*Sxa;
for k = 1:nxa+1
    if k == 1
        X(:,k) = xa;
    else
        X(:,k) = xa + c(k-1,:)';
```

```
        X(:,k+nxa) = xa - c(k-1,:)';
    end
end

% Time Update
Xx = zeros(nx, n);
x_bar = zeros(nx, 1);
for k = 1:n

% Gyro Model  w_bi(true) = w_bi(measured) - bias - noise
w_bi_b = [p-X(5,k)-X(8,k); q-X(6,k)-X(9,k); r-X(7, k)-X(10,k)];

if SIM == 1
    % No Sidereal Rate
    w_bt = w_bi_b;
else
    % Sidereal Rate Included
    R_t2b = rot_t2b(X(1:4,k));
    w_bt = w_bi_b - R_t2b*[wei*cos(lat0); 0; -wei*sin(lat0)];
end

% Quaternion Update
w1 = w_bt(1); w2 = w_bt(2); w3 = w_bt(3);
Q_kp1 = X(1:4, k) + (dt/2)*[0,  -w1,  -w2,  -w3;
                        w1,  0,    w3,  -w2;
                        w2, -w3,   0,    w1;
                        w3,  w2,  -w1,   0]*X(1:4, k);

% Quaternion Normalization
Q_kp1_n = Q_kp1/sqrt(Q_kp1'*Q_kp1);

% Gyro Bias Update
if SIM == 1
    % Constant Bias
    bg = X(5:7,k);
else
    % Random Walk Bias
    bg = X(5:7,k) + dt*X(11:13,k);
end

% Rebuild the State Vector for Each Iteration
Xx(:,k) = [Q_kp1_n; bg];

% Calculated the Weighted Mean of the State Vector
x_bar = x_bar + wm(k)*Xx(:,k);
end

% Calculate Error Covariance
Ax = sqrt(wc(2))*(Xx(:,2:n) - x_bar(:,ones(1,n-1)));
% QR Decomposition
[~, sx] = qr(Ax',0);
% Cholesky Update (Downdate for negative zeroth weight)
Sx = cholupdate(sx, sqrt(-wc(1))*(Xx(:,1)-x_bar), '-');
```

```
% Non-Linear Measurement Process Equations
Y = zeros(nR, n);                                   % Pre-fill
y_bar = zeros(nR,1);                                % Pre-fill
Heading = zeros(1,n);                               % Pre-fill
if isempty(H1)
    H1 = zeros(1,n);                                % Pre-fill
    H2 = zeros(1,n);                                % Pre-fill
    Heading0 = zeros(1,n);                          % Pre-fill
    ii = zeros(1,n);                                % Pre-fill
end
for k = 1:n

% Rotation Matrix (LTP to Body)
R_t2b = rot_t2b(Xx(1:4,k));
% Gyro Measurements
w_bi = [p; q; r] - Xx(5:7,k);
if SIM == 1
    noise_R = X(11:14,k);
    w_bt = w_bi;
else
    noise_R = X(14:17,k);
    w_bt = w_bi - R_t2b*[wei*cos(lat0); 0; -wei*sin(lat0)];
end

% Quaternion to Euler Angles
q0 = Xx(1,k); q1 = Xx(2,k); q2 = Xx(3,k); q3 = Xx(4,k);
psi = mod(atan2(2*(q1*q2+q0*q3), q0^2+q1^2-q2^2-q3^2)+ noise_R(4),
2*pi);
% Unwrap Heading Angle From [0, 2*pi] Range to Avoid Jumps
if nn == 1
    H1(k) = psi;
    Heading0(k) = H1(k);
    Heading(k) = H1(k);
else
    H2(k) = psi;
    if (H2(k)-H1(k)) <= -100*pi/180
        ii(k) = ii(k)+1;
    end
    if (H2(k)-H1(k)) >= 100*pi/180
        ii(k) = ii(k)-1;
    end
    Heading(k) = Heading0(k)+((H2(k)+2*pi*ii(k)) - Heading0(k));
    H1(k) = H2(k);
end

% Accelerometer Estimate
FB_hat = R_t2b*[ax_gps; ay_gps; az_gps] + ...
        cross(w_bt, R_t2b*x_bar2(4:6)) - ...
        R_t2b*[0; 0; g]-x_bar2(7:9)-noise_R(1:3);

% Calculated Observations
Y(:,k)=[FB_hat; Heading(k)];
```

```
% Calculated Observation Mean
y_bar = y_bar + wm(k)*Y(:,k);
end

% Estimated Observation Covariance
Bx = sqrt(wc(2))*(Y(:,2:n) - y_bar(:,ones(1,n-1)));
% QR Decomposition
[~, sy] = qr(Bx',0); % **"qr" PRODUCES UPPER TRIANGULAR MATRIX**
% Cholesky Update (or Downdate)
Sy = cholupdate(sy, sqrt(-wc(1))*(Y(:,1)-y_bar), '-');
% ***Sy MUST BE LOWER TRIANGULAR***
Sy = Sy';

% Estimated Cross Covariance
Pxy = zeros(nx,nR);
for k = 1:n
    Pxy = Pxy + wc(k)*((Xx(:,k) - x_bar)*(Y(:,k) - y_bar)');
end

% Measurement Method 2 (Accelerometer)

% Heading Unwrap from [0, 2*pi]
if nn == 1
    H3 = heading;
    H4 = 0;
    psi0 = H3;
    psi_m = H3;
    jj=0;
else
    H4 = heading;
    if (H4-H3) <= -100*pi/180
        jj = jj+1;
    end
    if (H4-H3) >= 100*pi/180
        jj = jj-1;
    end
    psi_m = psi0+((H4+2*pi*jj) - psi0);
    H3 = H4;
end

% Measurement Vector
Z = [fx; fy; fz; psi_m];

% Kalman Filter Equations
K = (Pxy/Sy')/Sy;                                % Efficient Least
Squares
x_bar = x_bar + K*(Z - y_bar);                   % Kalman Correction

% Covariance Corrections
Ux = K*Sy;
for k = 1:nR
    Sx = cholupdate(Sx, Ux(:,k), '-');           % Cholesky Downdate
end
```

```
Sx = Sx';                                          % MUST BE LOWER
TRIANGULAR

nn = 2;
%% INS ESTIMATOR

% Build Augmented State Vector
X = zeros(nxa2, n2);
xa = [x_bar2; zeros(nxa2-nx2,1)];

% Build Augmented Covariance Matrix
Sxa2 = [Sx2 zeros(nx2,nQ2) zeros(nx2,nR2);
        zeros(nQ2,nx2) Qv2 zeros(nQ2,nR2);
        zeros(nR2,nx2) zeros(nR2,nQ2) R2];

% Construct n+2 SIGMA Vectors
c = eta2*Sxa2;
for k = 1:nxa2+1
    if k == 1
        X(:,k) = xa;
    else
        X(:,k) = xa + c(k-1,:)';
        X(:,k+nxa2) = xa - c(k-1,:)';
    end
end

% Time Update
Xx = zeros(nx2, n2);                               % Pre-fill
x_bar2 = zeros(nx2, 1);                            % Pre-fill
for k = 1:n2
% Rotation Matrix
R_t2b = rot_t2b(x_bar(1:4));
R_b2t = R_t2b';

% Position Update
P_kp1 = X(1:3,k) + dt*[X(4:5,k); -X(6,k)];

% Velocity and Bias Update
if SIM == 1
% No Sidereal
    V_kp1 = X(4:6,k) + dt*(R_b2t*([fx; fy; fz] - X(7:9,k) - ...
            X(10:12, k)) + [0; 0; g]);
    ba_kp1 = X(7:9,k);
else
% Sidereal Included
    V_kp1 = X(4:6,k) + dt*(R_b2t*([fx; fy; fz] - X(7:9,k) - ...
            X(10:12, k)) + [0; 0; g] - 2*cross([wei*cos(lat0); ...
            0; -wei*sin(lat0)], X(4:6,k)));
    ba_kp1 = X(7:9,k) + dt*X(13:15,k);
end

% States
Xx(:,k) = [P_kp1; V_kp1; ba_kp1];
```

```
% Calculated Mean
x_bar2 = x_bar2 + wm2(k)*Xx(:,k);
end

% Predicted Error Covariance
Ax = sqrt(wc2(2))*(Xx(:,2:n2) - x_bar2(:,ones(1,n2-1)));
% QR Decomposition
[~, Sx2] = qr(Ax',0);
% Cholesky Factor Update
Sx2 = cholupdate(Sx2, sqrt(-wc2(1))*(Xx(:,1)-x_bar2), '-');

% Measurement Equations
Y = zeros(nR2, n2);                                    % Pre-fill
y_bar = zeros(nR2,1);                                  % Pre-fill
for k = 1:n2

if SIM == 1
    % Calculated Observations
    Y(:,k)=Xx(1:6,k)+X(13:18,k);
else
    % Calculated Observations
    Y(:,k)=Xx(1:6,k)+X(16:21,k);
end

% Calculated Observation Mean
y_bar = y_bar + wm2(k)*Y(:,k);
end

% Estimated Observation Covariance
Bx = sqrt(wc2(2))*(Y(:,2:n2) - y_bar(:,ones(1,n2-1)));
% QR Decomposition
[~, Sy] = qr(Bx',0);
% Cholesky Factor Update
Sy = cholupdate(Sy, sqrt(-wc2(1))*(Y(:,1)-y_bar), '-');
% Make Sy Lower Triangular
Sy = Sy';

% Estimated Cross Covariance
Pxy = zeros(nx2,nR2);
for k = 1:n2
    Pxy = Pxy + wc2(k)*((Xx(:,k) - x_bar2)*(Y(:,k) - y_bar)');
end

% Measurements
Z = [N_sat; E_sat; D_sat; speed*cos(CoG); speed*sin(CoG); Vd_sat];

% Kalman Filter Equations
% Kalman Gain w/ Efficient Least Squares
K = (Pxy/Sy')/Sy;
% Kalman State Corrections
x_bar2 = x_bar2 + K*(Z - y_bar);
% Covariance Correction
Ux = K*Sy;
for k = 1:nR2
```

```
    Sx2 = cholupdate(Sx2, Ux(:,k), '-');
end
% Make Sx2 Lower Right Triangular
Sx2 = Sx2';

X_HAT = [x_bar(1:4); x_bar2(1:6); x_bar(5:7); x_bar2(7:9)];
end
kk=kk+1;
end

function R_t2b = rot_t2b(x)
q0 = x(1); q1 = x(2); q2 = x(3); q3 = x(4);
R_t2b = [q0^2+q1^2-q2^2-q3^2 2*(q1*q2+q0*q3) 2*(q1*q3-q0*q2);
         2*(q2*q1-q0*q3) q0^2-q1^2+q2^2-q3^2 2*(q2*q3+q0*q1);
         2*(q3*q1+q0*q2) 2*(q3*q2-q0*q1) q0^2-q1^2-q2^2+q3^2];
end

function q = e2q(phi, theta, psi)
% Form Rotation Matrix
R_psi = [cos(psi) sin(psi) 0; -sin(psi) cos(psi) 0; 0 0 1];
R_theta = [cos(theta) 0 -sin(theta); 0 1 0; sin(theta) 0 cos(theta)];
R_phi = [1 0 0; 0 cos(phi) sin(phi); 0 -sin(phi) cos(phi)];
R_n2b = R_phi*R_theta*R_psi;

% Extract Quaternions
q0=sqrt(1+trace(R_n2b))/2;
q1=(R_n2b(2,3)-R_n2b(3,2))/(4*q0);
q2=(R_n2b(3,1)-R_n2b(1,3))/(4*q0);
q3=(R_n2b(1,2)-R_n2b(2,1))/(4*q0);

q = [q0; q1; q2; q3];
end
```

E. SPHERICAL SIMPLEX UNSCENTED KALMAN FILTER MATLAB CODE

1. Measurement Model One Implementation

```
function X_HAT = SSUKF1(u)
%% SPHERICAL SIMPLEX UNSCENTED KALMAN FILTER
% Measurement Model 1
% LT Steven Terjesen
% September 2014

% This estimator is built for use in MATLAB Simulink. There are 38
inputs
% required and can be run in Simulink with an "Interpolated MATLAB
% Function" block. The first 13 inputs are the vehicle data: Course
Over
% Ground, Speed Over Ground, Accelerometer Measurements, Gyro
Measurements,
```

```
% GPS Measurements (In LTP XNorth-YEast-ZDown), Heading Measurement,
and
% Vertical Velocity (zeroed out for SEAFOXII data). Inputs 14 through
34
% are the Process Noise and Measurement Noise diagonal elements. These
% elements are not hard coded to allow for easier tuning. Inputs 35
% through 37 are the N-E-D accelerations in the LTP frame. Input 38 is
a
% toggle for selecting whether the data is from Condor or SEAFOX II.

%% System Inputs
% Measurement inputs
CoG = u(1);                                          %GPS Course Over Ground
speed = u(2);                                        %GPS Speed Over Ground
fx = u(3); fy = u(4); fz = u(5);                     %IMU Accelerations
p = u(6); q = u(7); r = u(8);                        %IMU Angular Rates
N_sat = u(9); E_sat = u(10); D_sat = u(11);          %GPS Position (NED)
heading = u(12);                                     %GPS Heading
Vd_sat = u(13);                                      %GPS LTP Down Velocity

% Process and Measurement Noise Matrices
R = diag(u(14:16));                                  %AHRS Measurement Noise
R2 = diag(u(17:22));                                 %INS Measurement Noise

SIM = u(38);                                         %Condor(1) or
SEAFOXII(2)
%                                                     Simulation Selector

if SIM == 1
    Qv = diag(u(23:25));                             %AHRS Process Noise
Condor
    Qv2 = diag(u(29:31));                            %INS Process Noise
Condor
else
    Qv = diag(u(23:28));                             %AHRS Process Noise
SEAFOX
    Qv2 = diag(u(29:34));                            %INS Process Noise
SEAFOX
    Vd_sat = 0;                                      %No Vertical Velocity
%                                                     measurement available
on
%                                                     SEAFOX II, set Vd=0.
    D_sat = 0;                                       %No Altitude
Measurement
%                                                     available on SEAFOXII,
set
%                                                     D_sat = 0;
end

% GPS Accelerations
ax_gps = u(35); ay_gps = u(36); az_gps = u(37); %GPS Accelerations as
                                                % calculated from 3rd
                                                % Order Filter
```

```
%% Initialization
persistent x_bar Heading0 H1 H2 ii jj psi0 H3 H4 ...
Px Px2 kk dt g nn nx nxa nQ nR n wc wm x_bar2 nx2 nxa2 n2 nQ2 nR2 wc2
...
wm2 sig sig2 wei lat0

% Allows time for Condor To steady out during live testing
if isempty(kk)
    kk = 0;
end
if kk < 1 && SIM==1
    X_HAT = zeros(16,1);
else

if isempty(x_bar)
% Miscellaneous
dt = 0.01;                                          %Filter dt [sec]
g = 9.8;                                            %Gravity Constant
[m/s^2]
wei = 7.292115*10^-5;                               %Sidereal Rate [rad/s]
lat0= 0.639268394832413;                            %Origin of LTP [rad]
%lon0 = -2.115435878466264
nn=1;                                               %Heading Count
Initializer

% AHRS Initialization
% Euler Angle Initialization
rx = ax_gps*cos(heading)+ay_gps*sin(heading);
ry = -ax_gps*sin(heading)+ay_gps*cos(heading);
rz = az_gps-g;
theta_0 = atan((-rx*rz - fx*sqrt(rx^2+rz^2-fx^2))/(fx^2-rz^2));
r_theta = rx*sin(theta_0) + rz*cos(theta_0);
fc = fy-speed*r;
phi_0 = atan((r_theta*ry + fc*sqrt(ry^2+r_theta^2-fc^2))/(fc^2-
r_theta^2));
psi_0=heading;
% Euler Angles to Quaternions
q_0 = e2q(phi_0, theta_0, psi_0);
% Gyro Bias Initial Values
bg_0 = [0; 0; 0];
% Inital State Vector
x_bar = [q_0; bg_0];
% Initial Covariance Estimate
Px = diag([1e-5*ones(1,4) 1e-3*ones(1,3)]);

%INS Initialization
% Position & Velocity Initialization
x_0 = N_sat;
y_0 = E_sat;
z_0 = D_sat;
vn_0 = speed*cos(CoG);
ve_0 = speed*sin(CoG);
vd_0 = Vd_sat;
% Accelerometer Initial Bias Estimate
ba_0 = [0; 0; 0];
```

```
% Initial State Vector
x_bar2 = [x_0; y_0; z_0; vn_0; ve_0; vd_0; ba_0];
% Initial Covariance Estimate
Px2 = diag([1e-5*ones(1,3), 1e-5*ones(1,3), 1e-3*ones(1,3)]);

% AHRS SIGMA Weights Initialization
nx = length(x_bar);                                % Number of AHRS States
nQ = length(diag(Qv));                             % Process Noise States
nR = length(diag(R));                              % Measurement Noise
States
nxa = nx+nQ+nR;                                    % Total Augmented
States
n = nxa+2;                                         % Number of Iterations
alpha = 1e-3;                                      % Tunable Scaling
Factor
beta = 2;                                          % beta = 2 for Gaussian
PDF
Wx0 = 1/3;                                         % Tunable, 0<W<1
Wx = zeros(n,1);                                   % Pre-fill
% Simplex Weights
for k = 1:n
    if k == 1
        Wx(k) = Wx0;
    else
        Wx(k) = (1-Wx0)/(nxa+1);
    end
end
% Scaled Simplex Weights
for k = 1:n
    if k == 1
        wx(k)= 1+(Wx(k)-1)/alpha^2;
    else
        wx(k) = Wx(k)/alpha^2;
    end
end
% Incorporate prior PDF knowledge for Higher Order Moments
for k = 1:n
    if k == 1
        wm(k) = wx(k);
        wc(k) = wx(k)+(1-alpha^2+beta);
    else
        wm(k) = wx(k);
        wc(k) = wx(k);
    end
end

%Sigma Matrix
sig = zeros(nxa, n);                               % Pre-fill
sig(1,1) = 0;                                      % Initialize (1,1)
sig(1,2) = -1/sqrt(2*wx(2));                       % Initialize (1,2)
sig(1,3) = 1/sqrt(2*wx(2));                        % Initialize (1,3)
for j = 2:nxa
    for i = 1:1
        sig(j,i) = 0;
    end
```

```
    for i = 2:j+1
        sig(j,i) = -1/sqrt(j*(j+1)*wx(2));
    end
    for i = j+2:j+2
        sig(j,i) = j/sqrt(j*(j+1)*wx(2));
    end
end

% INS SIGMA Weights Initialization
nx2 = length(x_bar2);                           % Number of INS States
nQ2 = length(diag(Qv2));                        % Process Noise States
nR2 = length(diag(R2));                         % Measurement Noise
States
nxa2 = nx2+nQ2+nR2;                             % Total Augmented
States
n2 = nxa2+2;                                    % Total Iterations
alpha = 1e-3;                                   % Tunable, 0<alpha<1
beta = 2;                                       % beta = 2 for Gaussian
PDF
Wx02 = 1/3;                                     % Tunable, 0<W<1
Wx2 = zeros(n2,1);                              % Pre-fill

% Sigma Weights
for k = 1:n2
    if k == 1
        Wx2(k) = Wx02;
    else
        Wx2(k) = (1-Wx02)/(nxa2+1);
    end
end

% Scaled Sigma Weights
for k = 1:n2
    if k == 1
        wx2(k)= 1+(Wx2(k)-1)/alpha^2;
    else
        wx2(k) = Wx2(k)/alpha^2;
    end
end

% Incorporate prior PDF knowledge for Higher Order Moments
for k = 1:n2
    if k == 1
        wm2(k) = wx2(k);
        wc2(k) = wx2(k)+(1-alpha^2+beta);
    else
        wm2(k) = wx2(k);
        wc2(k) = wx2(k);
    end
end

% Sigma Matrix
sig2 = zeros(nxa2, n2);                         % Pre-fill
sig2(1,1) = 0;                                  % Initialize (1,1)
```

```
sig2(1,2) = -1/sqrt(2*wx2(2));                  % Initialize (1,2)
sig2(1,3) = 1/sqrt(2*wx2(2));                   % Initialize (1,3)
for j = 2:nxa2
    for i = 1:1
        sig2(j,i) = 0;
    end
    for i = 2:j+1
        sig2(j,i) = -1/sqrt(j*(j+1)*wx2(2));
    end
    for i = j+2:j+2
        sig2(j,i) = j/sqrt(j*(j+1)*wx2(2));
    end
end

end
%% AHRS ESTIMATOR

% Build Augmented State Vector
X = zeros(nxa, n);
xa = [x_bar; zeros(nxa-nx,1)];

% Build Augmented Covariance Matrix
Pxa = [Px zeros(nx,nQ) zeros(nx,nR);
       zeros(nQ,nx) Qv zeros(nQ,nR);
       zeros(nR,nx) zeros(nR,nQ) R];

% Build n+2 Sigma Vectors
for k = 1:n
    X(:,k) = xa + Pxa*sig(:,k);
end

% Time Update
Xx = zeros(nx, n);
x_bar = zeros(nx, 1);
for k = 1:n

% Gyro Model  w_bi(true) = w_bi(measured) - bias - noise
w_bi_b = [p-X(5,k)-X(8,k); q-X(6,k)-X(9,k); r-X(7, k)-X(10,k)];

if SIM == 1
    % No Sidereal Rate
    w_bt = w_bi_b;
else
    % Sidereal Rate Included
    R_t2b = rot_t2b(X(1:4,k));
    w_bt = w_bi_b - R_t2b*[wei*cos(lat0); 0; -wei*sin(lat0)];
end

% Quaternion Update
w1 = w_bt(1); w2 = w_bt(2); w3 = w_bt(3);
Q_kp1 = X(1:4, k) + (dt/2)*[0,  -w1,  -w2,  -w3;
                        w1,  0,    w3,  -w2;
                        w2, -w3,   0,    w1;
                        w3,  w2,  -w1,   0]*X(1:4, k);
```

```
% Quaternion Normalization
Q_kp1_n = Q_kp1/sqrt(Q_kp1'*Q_kp1);

% Gyro Bias Update
if SIM == 1
    % Constant Bias
    bg = X(5:7,k);
else
    % Random Walk Bias
    bg = X(5:7,k) + dt*X(11:13,k);
end

% Rebuild the State Vector for Each Iteration
Xx(:,k) = [Q_kp1_n; bg];

% Calculated the Weighted Mean of the State Vector
x_bar = x_bar + wm(k)*Xx(:,k);
end

% Calculate Error Covariance
Px= zeros(nx, nx);
for k = 1:n
    Px = Px + wc(k)*((Xx(:,k) - x_bar)*(Xx(:,k) - x_bar)');
end

% Non-Linear Measurement Process Equations
Y = zeros(nR, n);                                % Pre-fill
y_bar = zeros(nR,1);                             % Pre-fill
Heading = zeros(1,n);                            % Pre-fill
if isempty(H1)
    H1 = zeros(1,n);                             % Pre-fill
    H2 = zeros(1,n);                             % Pre-fill
    Heading0 = zeros(1,n);                       % Pre-fill
    ii = zeros(1,n);                             % Pre-fill
end
for k = 1:n

if SIM == 1
    noise_R = X(11:13,k);
else
    noise_R = X(14:16,k);
end

% Quaternion to Euler Angles
q0 = Xx(1,k); q1 = Xx(2,k); q2 = Xx(3,k); q3 = Xx(4,k);
phi = atan2(2*(q2*q3+q0*q1), q0^2-q1^2-q2^2+q3^2) + noise_R(1);
theta = asin(-2*(q1*q3-q0*q2))+ noise_R(2);
psi = mod(atan2(2*(q1*q2+q0*q3), q0^2+q1^2-q2^2-q3^2)+ noise_R(3),
2*pi);

% Unwrap Heading Angle From [0, 2*pi] Range to Avoid Jumps
if nn == 1
    H1(k) = psi;
```

```
    Heading0(k) = H1(k);
    Heading(k) = H1(k);
else
    H2(k) = psi;
    if (H2(k)-H1(k)) <= -100*pi/180
        ii(k) = ii(k)+1;
    end
    if (H2(k)-H1(k)) >= 100*pi/180
        ii(k) = ii(k)-1;
    end
    Heading(k) = Heading0(k)+((H2(k)+2*pi*ii(k)) - Heading0(k));
    H1(k) = H2(k);
end

% Calculated Observations
Y(:,k)=[phi; theta; Heading(k)];

% Calculated Observation Mean
y_bar = y_bar + wm(k)*Y(:,k);
end

% Estimated Observation Covariance
Py = zeros(nR,nR);
for k = 1:n
    Py = Py + wc(k)*((Y(:,k) - y_bar)*(Y(:,k) - y_bar)');
end

% Estimated Cross Covariance
Pxy = zeros(nx,nR);
for k = 1:n
    Pxy = Pxy + wc(k)*((Xx(:,k) - x_bar)*(Y(:,k) - y_bar)');
end

% Measurement Method 1 (Accelerometer-GPS-Compass Fusion)
% Rotate GPS XYZ Acceleration about Heading
rx = ax_gps*cos(heading)+ay_gps*sin(heading);
ry = -ax_gps*sin(heading)+ay_gps*cos(heading);
rz = az_gps-g;
% Computer Pitch
theta = atan((-rx*rz - fx*sqrt(rx^2+rz^2-fx^2)) / (fx^2-rz^2));
% Compute Roll
r_theta = rx*sin(theta) + rz*cos(theta);
fc = fy-(norm(x_bar2(4:6)))*r; % Added Coriolis Term
phi = atan((r_theta*ry + fc*sqrt(ry^2+r_theta^2-fc^2)) / (fc^2-
r_theta^2));

% Heading Unwrap from [0, 2*pi]
if nn == 1
    H3 = heading;
    H4 = 0;
    psi0 = H3;
    psi_m = H3;
    jj=0;
else
```

```
    H4 = heading;
    if (H4-H3) <= -100*pi/180
        jj = jj+1;
    end
    if (H4-H3) >= 100*pi/180
        jj = jj-1;
    end
    psi_m = psi0+((H4+2*pi*jj) - psi0);
    H3 = H4;
end

% Measurement Vector
Z = [phi; theta; psi_m];

% Kalman Filter Equations
K = Pxy/Py;                                    % Kalman Gain
x_bar = x_bar + K*(Z - y_bar);                 % Kalman Correction

% Covariance Corrections
Px = Px - K*Py*K';

nn = 2;
%% INS ESTIMATOR

% Build Augmented State Vector
X = zeros(nxa2, n2);
xa = [x_bar2; zeros(nxa2-nx2,1)];

% Build Augmented Covariance Matrix
Pxa2 = [Px2 zeros(nx2,nQ2) zeros(nx2,nR2);
        zeros(nQ2,nx2) Qv2 zeros(nQ2,nR2);
        zeros(nR2,nx2) zeros(nR2,nQ2) R2];

% Construct n+2 SIGMA Vectors
for k = 1:n2
    X(:,k) = xa + Pxa2*sig2(:,k);
end

% Time Update
Xx = zeros(nx2, n2);                               % Pre-fill
x_bar2 = zeros(nx2, 1);                            % Pre-fill
for k = 1:n2
% Rotation Matrix
R_t2b = rot_t2b(x_bar(1:4));
R_b2t = R_t2b';

% Position Update
P_kp1 = X(1:3,k) + dt*[X(4:5,k); -X(6,k)];

% Velocity and Bias Update
if SIM == 1
% No Sidereal
    V_kp1 = X(4:6,k) + dt*(R_b2t*([fx; fy; fz] - X(7:9,k) - ...
```

```
            X(10:12, k)) + [0; 0; g]);
    ba_kp1 = X(7:9,k);
else
% Sidereal Included
    V_kp1 = X(4:6,k) + dt*(R_b2t*([fx; fy; fz] - X(7:9,k) - ...
            X(10:12, k)) + [0; 0; g] - 2*cross([wei*cos(lat0); ...
            0; -wei*sin(lat0)], X(4:6,k)));
    ba_kp1 = X(7:9,k) + dt*X(13:15,k);
end

% States
Xx(:,k) = [P_kp1; V_kp1; ba_kp1];

% Calculated Mean
x_bar2 = x_bar2 + wm2(k)*Xx(:,k);
end

% Predicted Error Covariance
Px2= zeros(nx2, nx2);
for k = 1:n2
    Px2 = Px2 + wc2(k)*((Xx(:,k) - x_bar2)*(Xx(:,k) - x_bar2)');
end

% Measurement Equations
Y = zeros(nR2, n2);                                 % Pre-fill
y_bar = zeros(nR2,1);                               % Pre-fill
for k = 1:n2

if SIM == 1
    % Calculated Observations
    Y(:,k)=Xx(1:6,k)+X(13:18,k);
else
    % Calculated Observations
    Y(:,k)=Xx(1:6,k)+X(16:21,k);
end

% Calculated Observation Mean
y_bar = y_bar + wm2(k)*Y(:,k);
end

% Estimated Observation Covariance
Py = zeros(nR2,nR2);
for k = 1:n2
    Py = Py + wc2(k)*((Y(:,k) - y_bar)*(Y(:,k) - y_bar)');
end

% Estimated Cross Covariance
Pxy = zeros(nx2,nR2);
for k = 1:n2
    Pxy = Pxy + wc2(k)*((Xx(:,k) - x_bar2)*(Y(:,k) - y_bar)');
end

% Measurements
Z = [N_sat; E_sat; D_sat; speed*cos(CoG); speed*sin(CoG); Vd_sat];
```

```
% Kalman Filter Equations
K = Pxy/Py;                                          % Kalman Gain
% Kalman State Corrections
x_bar2 = x_bar2 + K*(Z - y_bar);
% Covariance Correction
Px2 = Px2 - K*Py*K';

X_HAT = [x_bar(1:4); x_bar2(1:6); x_bar(5:7); x_bar2(7:9)];
end
kk=kk+1;

end
function R_t2b = rot_t2b(x)
q0 = x(1); q1 = x(2); q2 = x(3); q3 = x(4);
R_t2b = [q0^2+q1^2-q2^2-q3^2 2*(q1*q2+q0*q3) 2*(q1*q3-q0*q2);
         2*(q2*q1-q0*q3) q0^2-q1^2+q2^2-q3^2 2*(q2*q3+q0*q1);
         2*(q3*q1+q0*q2) 2*(q3*q2-q0*q1) q0^2-q1^2-q2^2+q3^2];
end

function q = e2q(phi, theta, psi)
% Form Rotation Matrix
R_psi = [cos(psi) sin(psi) 0; -sin(psi) cos(psi) 0; 0 0 1];
R_theta = [cos(theta) 0 -sin(theta); 0 1 0; sin(theta) 0 cos(theta)];
R_phi = [1 0 0; 0 cos(phi) sin(phi); 0 -sin(phi) cos(phi)];
R_n2b = R_phi*R_theta*R_psi;

% Extract Quaternions
q0=sqrt(1+trace(R_n2b))/2;
q1=(R_n2b(2,3)-R_n2b(3,2))/(4*q0);
q2=(R_n2b(3,1)-R_n2b(1,3))/(4*q0);
q3=(R_n2b(1,2)-R_n2b(2,1))/(4*q0);

q = [q0; q1; q2; q3];
end
```

2. Measurement Model Two Implementation

```
function X_HAT = SSUKF2(u)
%% SPHERICAL SIMPLEX UNSCENTED KALMAN FILTER
% Measurement Model 2
% LT Steven Terjesen
% September 2014

% This estimator is built for use in MATLAB Simulink. There are 39
inputs
% required and can be run in Simulink with an "Interpolated MATLAB
% Function" block. The first 13 inputs are the vehicle data: Course
Over
% Ground, Speed Over Ground, Accelerometer Measurements, Gyro
Measurements,
% GPS Measurements (In LTP XNorth-YEast-ZDown), Heading Measurement,
and
```

```
% Vertical Velocity (zeroed out for SEAFOXII data). Inputs 14 through
35
% are the Process Noise and Measurement Noise diagonal elements. These
% elements are not hard coded to allow for easier tuning. Inputs 36
% through 38 are the N-E-D accelerations in the LTP frame. Input 39 is
a
% toggle for selecting whether the data is from Condor or SEAFOX II.

%% System Inputs
% Measurement inputs
CoG = u(1);                                          %GPS Course Over Ground
speed = u(2);                                        %GPS Speed Over Ground
fx = u(3); fy = u(4); fz = u(5);                     %IMU Accelerations
p = u(6); q = u(7); r = u(8);                        %IMU Angular Rates
N_sat = u(9); E_sat = u(10); D_sat = u(11);          %GPS Position (NED)
heading = u(12);                                     %GPS Heading
Vd_sat = u(13);                                      %GPS LTP Down Velocity

% Process and Measurement Noise Matrices
R = diag(u(14:17));                                  %AHRS Measurement Noise
R2 = diag(u(18:23));                                 %INS Measurement Noise

SIM = u(39);                                         %Condor(1) or
SEAFOXII(2)
%                                                     Simulation Selector

if SIM == 1
    Qv = diag(u(24:26));                             %AHRS Process Noise
Condor
    Qv2 = diag(u(30:32));                            %INS Process Noise
Condor
else
    Qv = diag(u(24:29));                             %AHRS Process Noise
SEAFOX
    Qv2 = diag(u(30:35));                            %INS Process Noise
SEAFOX
    Vd_sat = 0;                                      %No Vertical Velocity
%                                                     measurement available
on
%                                                     SEAFOX II, set Vd=0.
    D_sat = 0;                                       %No Altitude
Measurement
%                                                     available on SEAFOXII,
set
%                                                     D_sat = 0;
end

% GPS Accelerations
ax_gps = u(36); ay_gps = u(37); az_gps = u(38); %GPS Accelerations as
                                                % calculated from 3rd
                                                % Order Filter

%% Initialization
persistent x_bar Heading0 H1 H2 ii jj psi0 H3 H4 ...
```

```
Px Px2 kk dt g nn nx nxa nQ nR n wc wm x_bar2 nx2 nxa2 n2 nQ2 nR2 wc2
...
wm2 sig sig2 wei lat0

% Allows time for Condor To steady out during live testing
if isempty(kk)
    kk = 0;
end
if kk < 1 && SIM==1
    X_HAT = zeros(16,1);
else

if isempty(x_bar)
% Miscellaneous
dt = 0.01;                                          %Filter dt [sec]
g = 9.8;                                            %Gravity Constant
[m/s^2]
wei = 7.292115*10^-5;                               %Sidereal Rate [rad/s]
lat0= 0.639268394832413;                            %Origin of LTP [rad]
%lon0 = -2.115435878466264
nn=1;                                               %Heading Count
Initializer

% AHRS Initialization
% Euler Angle Initialization
rx = ax_gps*cos(heading)+ay_gps*sin(heading);
ry = -ax_gps*sin(heading)+ay_gps*cos(heading);
rz = az_gps-g;
theta_0 = atan((-rx*rz - fx*sqrt(rx^2+rz^2-fx^2))/(fx^2-rz^2));
r_theta = rx*sin(theta_0) + rz*cos(theta_0);
fc = fy-speed*r;
phi_0 = atan((r_theta*ry + fc*sqrt(ry^2+r_theta^2-fc^2))/(fc^2-
r_theta^2));
psi_0=heading;
% Euler Angles to Quaternions
q_0 = e2q(phi_0, theta_0, psi_0);
% Gyro Bias Initial Values
bg_0 = [0; 0; 0];
% Inital State Vector
x_bar = [q_0; bg_0];
% Initial Covariance Estimate
Px = diag([1e-5*ones(1,4) 1e-3*ones(1,3)]);

%INS Initialization
% Position & Velocity Initialization
x_0 = N_sat;
y_0 = E_sat;
z_0 = D_sat;
vn_0 = speed*cos(CoG);
ve_0 = speed*sin(CoG);
vd_0 = Vd_sat;
% Accelerometer Initial Bias Estimate
ba_0 = [0; 0; 0];
% Initial State Vector
x_bar2 = [x_0; y_0; z_0; vn_0; ve_0; vd_0; ba_0];
```

```
% Initial Covariance Estimate
Px2 = diag([1e-5*ones(1,3), 1e-5*ones(1,3), 1e-3*ones(1,3)]);

% AHRS SIGMA Weights Initialization
nx = length(x_bar);                                % Number of AHRS States
nQ = length(diag(Qv));                             % Process Noise States
nR = length(diag(R));                              % Measurement Noise
States
nxa = nx+nQ+nR;                                    % Total Augmented
States
n = nxa+2;                                         % Number of Iterations
alpha = 1e-3;                                      % Tunable Scaleing
Factor
beta = 2;                                          % beta = 2 for Gaussian
PDF
Wx0 = 1/3;                                         % Tunable, 0<W<1
Wx = zeros(n,1);                                   % Pre-fill
% Simplex Weights
for k = 1:n
    if k == 1
        Wx(k) = Wx0;
    else
        Wx(k) = (1-Wx0)/(nxa+1);
    end
end
% Scaled Simplex Weights
for k = 1:n
    if k == 1
        wx(k)= 1+(Wx(k)-1)/alpha^2;
    else
        wx(k) = Wx(k)/alpha^2;
    end
end
% Incorperate prior PDF knowledge for Higher Order Moments
for k = 1:n
    if k == 1
        wm(k) = wx(k);
        wc(k) = wx(k)+(1-alpha^2+beta);
    else
        wm(k) = wx(k);
        wc(k) = wx(k);
    end
end

%Sigma Matrix
sig = zeros(nxa, n);                               % Pre-fill
sig(1,1) = 0;                                      % Initialize (1,1)
sig(1,2) = -1/sqrt(2*wx(2));                       % Initialize (1,2)
sig(1,3) = 1/sqrt(2*wx(2));                        % Initialize (1,3)
for j = 2:nxa
    for i = 1:1
        sig(j,i) = 0;
    end
    for i = 2:j+1
        sig(j,i) = -1/sqrt(j*(j+1)*wx(2));
```

```
    end
    for i = j+2:j+2
        sig(j,i) = j/sqrt(j*(j+1)*wx(2));
    end
end

% INS SIGMA Weights Initialization
nx2 = length(x_bar2);                           % Number of INS States
nQ2 = length(diag(Qv2));                        % Process Noise States
nR2 = length(diag(R2));                         % Measurement Noise
States
nxa2 = nx2+nQ2+nR2;                             % Total Augmented
States
n2 = nxa2+2;                                    % Total Iterations
alpha = 1e-3;                                   % Tunable, 0<alpha<1
beta = 2;                                       % beta = 2 for Gaussian
PDF
Wx02 = 1/3;                                     % Tunable, 0<W<1
Wx2 = zeros(n2,1);                              % Pre-fill

% Sigma Weights
for k = 1:n2
    if k == 1
        Wx2(k) = Wx02;
    else
        Wx2(k) = (1-Wx02)/(nxa2+1);
    end
end

% Scaled Sigma Weights
for k = 1:n2
    if k == 1
        wx2(k)= 1+(Wx2(k)-1)/alpha^2;
    else
        wx2(k) = Wx2(k)/alpha^2;
    end
end

% Incorperate prior PDF knowledge for Higher Order Moments
for k = 1:n2
    if k == 1
        wm2(k) = wx2(k);
        wc2(k) = wx2(k)+(1-alpha^2+beta);
    else
        wm2(k) = wx2(k);
        wc2(k) = wx2(k);
    end
end

% Sigma Matrix
sig2 = zeros(nxa2, n2);                         % Pre-fill
sig2(1,1) = 0;                                  % Initialize (1,1)
sig2(1,2) = -1/sqrt(2*wx2(2));                  % Initialize (1,2)
sig2(1,3) = 1/sqrt(2*wx2(2));                   % Initialize (1,3)
```

```
for j = 2:nxa2
    for i = 1:1
        sig2(j,i) = 0;
    end
    for i = 2:j+1
        sig2(j,i) = -1/sqrt(j*(j+1)*wx2(2));
    end
    for i = j+2:j+2
        sig2(j,i) = j/sqrt(j*(j+1)*wx2(2));
    end
end

end
%% AHRS ESTIMATOR

% Build Augmented State Vector
X = zeros(nxa, n);
xa = [x_bar; zeros(nxa-nx,1)];

% Build Augmented Covariance Matrix
Pxa = [Px zeros(nx,nQ) zeros(nx,nR);
       zeros(nQ,nx) Qv zeros(nQ,nR);
       zeros(nR,nx) zeros(nR,nQ) R];

% Build n+2 Sigma Vectors
for k = 1:n
    X(:,k) = xa + Pxa'*sig(:,k);
end

% Time Update
Xx = zeros(nx, n);
x_bar = zeros(nx, 1);
for k = 1:n

% Gyro Model  w_bi(true) = w_bi(measured) - bias - noise
w_bi_b = [p-X(5,k)-X(8,k); q-X(6,k)-X(9,k); r-X(7, k)-X(10,k)];

if SIM == 1
    % No Sidereal Rate
    w_bt = w_bi_b;
else
    % Sidereal Rate Included
    R_t2b = rot_t2b(X(1:4,k));
    w_bt = w_bi_b - R_t2b*[wei*cos(lat0); 0; -wei*sin(lat0)];
end

% Quaternion Update
w1 = w_bt(1); w2 = w_bt(2); w3 = w_bt(3);
Q_kp1 = X(1:4, k) + (dt/2)*[0,  -w1,  -w2,  -w3;
                            w1,  0,    w3,  -w2;
                            w2, -w3,   0,    w1;
                            w3,  w2,  -w1,   0]*X(1:4, k);

% Quaternion Normalization
```

```
Q_kp1_n = Q_kp1/sqrt(Q_kp1'*Q_kp1);

% Gyro Bias Update
if SIM == 1
    % Constant Bias
    bg = X(5:7,k);
else
    % Random Walk Bias
    bg = X(5:7,k) + dt*X(11:13,k);
end

% Rebuild the State Vector for Each Iteration
Xx(:,k) = [Q_kp1_n; bg];

% Calculated the Weighted Mean of the State Vector
x_bar = x_bar + wm(k)*Xx(:,k);
end

% Calculate Error Covariance
Px= zeros(nx, nx);
for k = 1:n
    Px = Px + wc(k)*((Xx(:,k) - x_bar)*(Xx(:,k) - x_bar)');
end

% Non-Linear Measurement Process Equations
Y = zeros(nR, n);                                % Pre-fill
y_bar = zeros(nR,1);                             % Pre-fill
Heading = zeros(1,n);                            % Pre-fill
if isempty(H1)
    H1 = zeros(1,n);                             % Pre-fill
    H2 = zeros(1,n);                             % Pre-fill
    Heading0 = zeros(1,n);                       % Pre-fill
    ii = zeros(1,n);                             % Pre-fill
end
for k = 1:n

% Rotation Matrix (LTP to Body)
R_t2b = rot_t2b(Xx(1:4,k));
% Gyro Measurements
w_bi = [p; q; r] - Xx(5:7,k);
if SIM == 1
    noise_R = X(11:14,k);
    w_bt = w_bi;
else
    noise_R = X(14:17,k);
    w_bt = w_bi - R_t2b*[wei*cos(lat0); 0; -wei*sin(lat0)];
end

% Quaternion to Euler Angles
q0 = Xx(1,k); q1 = Xx(2,k); q2 = Xx(3,k); q3 = Xx(4,k);
psi = mod(atan2(2*(q1*q2+q0*q3), q0^2+q1^2-q2^2-q3^2)+ noise_R(4),
2*pi);
% Unwrap Heading Angle From [0, 2*pi] Range to Aviod Jumps
if nn == 1
```

```
    H1(k) = psi;
    Heading0(k) = H1(k);
    Heading(k) = H1(k);
else
    H2(k) = psi;
    if (H2(k)-H1(k)) <= -100*pi/180
        ii(k) = ii(k)+1;
    end
    if (H2(k)-H1(k)) >= 100*pi/180
        ii(k) = ii(k)-1;
    end
    Heading(k) = Heading0(k)+((H2(k)+2*pi*ii(k)) - Heading0(k));
    H1(k) = H2(k);
end

% Accelerometer Estimate
FB_hat = R_t2b*[ax_gps; ay_gps; az_gps] + ...
        cross(w_bt, R_t2b*x_bar2(4:6)) - ...
        R_t2b*[0; 0; g]-x_bar2(7:9)-noise_R(1:3);

% Calculated Observations
Y(:,k)=[FB_hat; Heading(k)];

% Calculated Observation Mean
y_bar = y_bar + wm(k)*Y(:,k);
end

% Estimated Observation Covariance
Py = zeros(nR,nR);
for k = 1:n
    Py = Py + wc(k)*((Y(:,k) - y_bar)*(Y(:,k) - y_bar)');
end

% Estimated Cross Covariance
Pxy = zeros(nx,nR);
for k = 1:n
    Pxy = Pxy + wc(k)*((Xx(:,k) - x_bar)*(Y(:,k) - y_bar)');
end

% Measurement Method 2 (Accelerometer)

% Heading Unwrap from [0, 2*pi]
if nn == 1
    H3 = heading;
    H4 = 0;
    psi0 = H3;
    psi_m = H3;
    jj=0;
else
    H4 = heading;
    if (H4-H3) <= -100*pi/180
        jj = jj+1;
    end
    if (H4-H3) >= 100*pi/180
```

```
        jj = jj-1;
    end
    psi_m = psi0+((H4+2*pi*jj) - psi0);
    H3 = H4;
end

% Measurement Vector
Z = [fx; fy; fz; psi_m];

% Kalman Filter Equations
K = Pxy/Py;                                     % Kalman Gain
x_bar = x_bar + K*(Z - y_bar);                  % Kalman Correction

% Covariance Corrections
Px = Px - K*Py*K';

nn = 2;
%% INS ESTIMATOR

% Build Augmented State Vector
X = zeros(nxa2, n2);
xa = [x_bar2; zeros(nxa2-nx2,1)];

% Build Augmented Covariance Matrix
Pxa2 = [Px2 zeros(nx2,nQ2) zeros(nx2,nR2);
        zeros(nQ2,nx2) Qv2 zeros(nQ2,nR2);
        zeros(nR2,nx2) zeros(nR2,nQ2) R2];

% Construct n+2 SIGMA Vectors
for k = 1:n2
    X(:,k) = xa + Pxa2'*sig2(:,k);
end

% Time Update
Xx = zeros(nx2, n2);                                % Pre-fill
x_bar2 = zeros(nx2, 1);                             % Pre-fill
for k = 1:n2
% Rotation Matrix
R_t2b = rot_t2b(x_bar(1:4));
R_b2t = R_t2b';

% Position Update
P_kp1 = X(1:3,k) + dt*[X(4:5,k); -X(6,k)];

% Velocity and Bias Update
if SIM == 1
% No Sidereal
    V_kp1 = X(4:6,k) + dt*(R_b2t*([fx; fy; fz] - X(7:9,k) - ...
            X(10:12, k)) + [0; 0; g]);
    ba_kp1 = X(7:9,k);
else
% Sidereal Included
    V_kp1 = X(4:6,k) + dt*(R_b2t*([fx; fy; fz] - X(7:9,k) - ...
```

```
            X(10:12, k)) + [0; 0; g] - 2*cross([wei*cos(lat0); ...
            0; -wei*sin(lat0)], X(4:6,k)));
    ba_kp1 = X(7:9,k) + dt*X(13:15,k);
end

% States
Xx(:,k) = [P_kp1; V_kp1; ba_kp1];

% Calculated Mean
x_bar2 = x_bar2 + wm2(k)*Xx(:,k);
end

% Predicted Error Covariance
Px2= zeros(nx2, nx2);
for k = 1:n2
    Px2 = Px2 + wc2(k)*((Xx(:,k) - x_bar2)*(Xx(:,k) - x_bar2)');
end

% Measurement Equations
Y = zeros(nR2, n2);                                 % Pre-fill
y_bar = zeros(nR2,1);                               % Pre-fill
for k = 1:n2

if SIM == 1
    % Calculated Observations
    Y(:,k)=Xx(1:6,k)+X(13:18,k);
else
    % Calculated Observations
    Y(:,k)=Xx(1:6,k)+X(16:21,k);
end

% Calculated Observation Mean
y_bar = y_bar + wm2(k)*Y(:,k);
end

% Estimated Observation Covariance
Py = zeros(nR2,nR2);
for k = 1:n2
    Py = Py + wc2(k)*((Y(:,k) - y_bar)*(Y(:,k) - y_bar)');
end

% Estimated Cross Covariance
Pxy = zeros(nx2,nR2);
for k = 1:n2
    Pxy = Pxy + wc2(k)*((Xx(:,k) - x_bar2)*(Y(:,k) - y_bar)');
end

% Measurements
Z = [N_sat; E_sat; D_sat; speed*cos(CoG); speed*sin(CoG); Vd_sat];

% Kalman Filter Equations
K = Pxy/Py;                                         % Kalman Gain
% Kalman State Corrections
```

```
x_bar2 = x_bar2 + K*(Z - y_bar);
% Covariance Correction
Px2 = Px2 - K*Py*K';

X_HAT = [x_bar(1:4); x_bar2(1:6); x_bar(5:7); x_bar2(7:9)];
end
kk=kk+1;
end

function R_t2b = rot_t2b(x)
q0 = x(1); q1 = x(2); q2 = x(3); q3 = x(4);
R_t2b = [q0^2+q1^2-q2^2-q3^2 2*(q1*q2+q0*q3) 2*(q1*q3-q0*q2);
         2*(q2*q1-q0*q3) q0^2-q1^2+q2^2-q3^2 2*(q2*q3+q0*q1);
         2*(q3*q1+q0*q2) 2*(q3*q2-q0*q1) q0^2-q1^2-q2^2+q3^2];
end

function q = e2q(phi, theta, psi)
% Form Rotation Matrix
R_psi = [cos(psi) sin(psi) 0; -sin(psi) cos(psi) 0; 0 0 1];
R_theta = [cos(theta) 0 -sin(theta); 0 1 0; sin(theta) 0 cos(theta)];
R_phi = [1 0 0; 0 cos(phi) sin(phi); 0 -sin(phi) cos(phi)];
R_n2b = R_phi*R_theta*R_psi;

% Extract Quaternions
q0=sqrt(1+trace(R_n2b))/2;
q1=(R_n2b(2,3)-R_n2b(3,2))/(4*q0);
q2=(R_n2b(3,1)-R_n2b(1,3))/(4*q0);
q3=(R_n2b(1,2)-R_n2b(2,1))/(4*q0);

q = [q0; q1; q2; q3];
end
```

F. SQUARE ROOT SPHERICAL SIMPLEX UNSCENTED KALMAN FILTER MATLAB CODE

1. Measurement Model One Implementation

```
function X_HAT = SRSSUKF1(u)
%% SQUARE ROOOT SPHERICAL SIMPLEX UNSCENTED KALMAN FILTER
% Measurement Model 1
% LT Steven Terjesen
% September 2014

% This estimator is built for use in MATLAB Simulink. There are 38
inputs
% required and can be run in Simulink with an "Interpolated MATLAB
% Function" block. The first 13 inputs are the vehicle data: Course
Over
% Ground, Speed Over Ground, Accelerometer Measurements, Gyro
Measurements,
% GPS Measurements (In LTP XNorth-YEast-ZDown), Heading Measurement,
and
```

```
% Vertical Velocity (zeroed out for SEAFOXII data). Inputs 14 through
34
% are the Process Noise and Measurement Noise diagonal elements. These
% elements are not hard coded to allow for easier tuning. Inputs 35
% through 37 are the N-E-D accelerations in the LTP frame. Input 38 is
a
% toggle for selecting whether the data is from Condor or SEAFOX II.

%% System Inputs
% Measurement inputs
CoG = u(1);                                         %GPS Course Over Ground
speed = u(2);                                       %GPS Speed Over Ground
fx = u(3); fy = u(4); fz = u(5);                    %IMU Accelerations
p = u(6); q = u(7); r = u(8);                       %IMU Angular Rates
N_sat = u(9); E_sat = u(10); D_sat = u(11);         %GPS Position (NED)
heading = u(12);                                    %GPS Heading
Vd_sat = u(13);                                     %GPS LTP Down Velocity

% Process and Measurement Noise Matrices
R = chol(diag(u(14:16)));                           %AHRS Measurement Noise
R2 = chol(diag(u(17:22)));                          %INS Measurement Noise

SIM = u(38);                                        %Condor(1) or
SEAFOXII(2)

if SIM == 1
    Qv = chol(diag(u(23:25)));                      %AHRS Process Noise
Condor
    Qv2 = chol(diag(u(29:31)));                     %INS Process Noise
Condor
else
    Qv = chol(diag(u(23:28)));                      %AHRS Process Noise
SEAFOX
    Qv2 = chol(diag(u(29:34)));                     %INS Process Noise
SEAFOX
    Vd_sat = 0;                                     %No Vertical Velocity
%                                                    measurement available
on
%                                                    SEAFOX II, set Vd=0.
    D_sat = 0;                                      %No Altitude
Measurement
%                                                    available on SEAFOXII,
set
%                                                    D_sat = 0;
end

% GPS Accelerations
ax_gps = u(35); ay_gps = u(36); az_gps = u(37); %GPS Accelerations as
                                                % calculated from 3rd
                                                % Order Filter

%% Initialization
persistent x_bar Heading0 H1 H2 ii jj psi0 H3 H4 ...
```

```
Sx Sx2 kk dt g nn nx nxa nQ nR n wc wm x_bar2 nx2 nxa2 n2 nQ2 nR2 wc2
...
wm2 sig sig2 wei lat0

% Allows time for Condor To steady out during live testing
if isempty(kk)
    kk = 0;
end
if kk < 1 && SIM==1
    X_HAT = zeros(16,1);
else

if isempty(x_bar)
% Miscellaneous
dt = 0.01;                                          %Filter dt [sec]
g = 9.8;                                            %Gravity Constant
[m/s^2]
wei = 7.292115*10^-5;                               %Sidereal Rate [rad/s]
lat0= 0.639268394832413;                            %Origing of LTP [rad]
%lon0 = -2.115435878466264
nn=1;                                               %Heading Count
Initializer

% AHRS Initialization
% Euler Angle Initialization
rx = ax_gps*cos(heading)+ay_gps*sin(heading);
ry = -ax_gps*sin(heading)+ay_gps*cos(heading);
rz = az_gps-g;
theta_0 = atan((-rx*rz - fx*sqrt(rx^2+rz^2-fx^2))/(fx^2-rz^2));
r_theta = rx*sin(theta_0) + rz*cos(theta_0);
fc = fy-speed*r;
phi_0 = atan((r_theta*ry + fc*sqrt(ry^2+r_theta^2-fc^2))/(fc^2-
r_theta^2));
psi_0=heading;
% Euler Angles to Quaternions
q_0 = e2q(phi_0, theta_0, psi_0);
% Gyro Bias Initial Values
bg_0 = [0; 0; 0];
% Inital State Vector
x_bar = [q_0; bg_0];
% Initial Covariance Estimate
Px = diag([1e-6*ones(1,4) 1e-3*ones(1,3)]);
% Inital Matrix Square Root
Sx = chol(Px);

%INS Initialization
% Position & Velocity Initialization
x_0 = N_sat;
y_0 = E_sat;
z_0 = D_sat;
vn_0 = speed*cos(CoG);
ve_0 = speed*sin(CoG);
vd_0 = Vd_sat;
% Accelerometer Initial Bias Estimate
ba_0 = [0; 0; 0];
```

```
% Initial State Vector
x_bar2 = [x_0; y_0; z_0; vn_0; ve_0; vd_0; ba_0];
% Initial Covariance Estimate
Px2 = diag([1e-5*ones(1,3), 1e-5*ones(1,3), 1e-3*ones(1,3)]);
% Inital Matrix Square Root
Sx2 = chol(Px2);

% AHRS SIGMA Weights Initialization
nx = length(x_bar);                             % Number of AHRS States
nQ = length(diag(Qv));                          % Process Noise States
nR = length(diag(R));                           % Measurement Noise
States
nxa = nx+nQ+nR;                                 % Total Augmented
States
n = nxa+2;                                      % Number of Iterations
alpha = 1e-3;                                   % Tunable Scaling
Factor
beta = 2;                                       % beta = 2 for Gaussian
PDF
Wx0 = 1/3;                                      % Tunable, 0<W<1
Wx = zeros(n,1);                                % Pre-fill
% Simplex Weights
for k = 1:n
    if k == 1
        Wx(k) = Wx0;
    else
        Wx(k) = (1-Wx0)/(nxa+1);
    end
end
% Scaled Simplex Weights
for k = 1:n
    if k == 1
        wx(k)= 1+(Wx(k)-1)/alpha^2;
    else
        wx(k) = Wx(k)/alpha^2;
    end
end
% Incorperate prior PDF knowledge for Higher Order Moments
for k = 1:n
    if k == 1
        wm(k) = wx(k);
        wc(k) = wx(k)+(1-alpha^2+beta);
    else
        wm(k) = wx(k);
        wc(k) = wx(k);
    end
end

%Sigma Matrix
sig = zeros(nxa, n);                            % Pre-fill
sig(1,1) = 0;                                   % Initialize (1,1)
sig(1,2) = -1/sqrt(2*wx(2));                    % Initialize (1,2)
sig(1,3) = 1/sqrt(2*wx(2));                     % Initialize (1,3)
for j = 2:nxa
    for i = 1:1
```

```
        sig(j,i) = 0;
    end
    for i = 2:j+1
        sig(j,i) = -1/sqrt(j*(j+1)*wx(2));
    end
    for i = j+2:j+2
        sig(j,i) = j/sqrt(j*(j+1)*wx(2));
    end
end

% INS SIGMA Weights Initialization
nx2 = length(x_bar2);                           % Number of INS States
nQ2 = length(diag(Qv2));                        % Process Noise States
nR2 = length(diag(R2));                         % Measurement Noise
States
nxa2 = nx2+nQ2+nR2;                             % Total Augmented
States
n2 = nxa2+2;                                    % Total Iterations
alpha = 1e-3;                                   % Tunable, 0<alpha<1
beta = 2;                                       % beta = 2 for Gaussian
PDF
Wx02 = 1/3;                                     % Tunable, 0<W<1
Wx2 = zeros(n2,1);                              % Pre-fill

% Sigma Weights
for k = 1:n2
    if k == 1
        Wx2(k) = Wx02;
    else
        Wx2(k) = (1-Wx02)/(nxa2+1);
    end
end

% Scaled Sigma Weights
for k = 1:n2
    if k == 1
        wx2(k)= 1+(Wx2(k)-1)/alpha^2;
    else
        wx2(k) = Wx2(k)/alpha^2;
    end
end

% Incorperate prior PDF knowledge for Higher Order Moments
for k = 1:n2
    if k == 1
        wm2(k) = wx2(k);
        wc2(k) = wx2(k)+(1-alpha^2+beta);
    else
        wm2(k) = wx2(k);
        wc2(k) = wx2(k);
    end
end

% Sigma Matrix
```

```
sig2 = zeros(nxa2, n2);                          % Pre-fill
sig2(1,1) = 0;                                   % Initialize (1,1)
sig2(1,2) = -1/sqrt(2*wx2(2));                   % Initialize (1,2)
sig2(1,3) = 1/sqrt(2*wx2(2));                    % Initialize (1,3)
for j = 2:nxa2
    for i = 1:1
        sig2(j,i) = 0;
    end
    for i = 2:j+1
        sig2(j,i) = -1/sqrt(j*(j+1)*wx2(2));
    end
    for i = j+2:j+2
        sig2(j,i) = j/sqrt(j*(j+1)*wx2(2));
    end
end

end
%% AHRS ESTIMATOR

% Build Augmented State Vector
X = zeros(nxa, n);
xa = [x_bar; zeros(nxa-nx,1)];

% Build Augmented Covariance Matrix
Sxa = [Sx zeros(nx,nQ) zeros(nx,nR);
        zeros(nQ,nx) Qv zeros(nQ,nR);
        zeros(nR,nx) zeros(nR,nQ) R];

% Build n+2 Sigma Vectors
for k = 1:n
    X(:,k) = xa + Sxa*sig(:,k);
end

% Time Update
Xx = zeros(nx, n);
x_bar = zeros(nx, 1);
for k = 1:n

% Gyro Model  w_bi(true) = w_bi(measured) - bias - noise
w_bi_b = [p-X(5,k)-X(8,k); q-X(6,k)-X(9,k); r-X(7, k)-X(10,k)];

if SIM == 1
    % No Sidereal Rate
    w_bt = w_bi_b;
else
    % Sidereal Rate Included
    R_t2b = rot_t2b(X(1:4,k));
    w_bt = w_bi_b - R_t2b*[wei*cos(lat0); 0; -wei*sin(lat0)];
end

% Quaternion Update
w1 = w_bt(1); w2 = w_bt(2); w3 = w_bt(3);
Q_kp1 = X(1:4, k) + (dt/2)*[0,  -w1,  -w2,  -w3;
                        w1,  0,    w3,  -w2;
```

```
                        w2, -w3,   0,    w1;
                        w3,  w2,  -w1,   0]*X(1:4, k);

% Quaternion Normalization
Q_kp1_n = Q_kp1/sqrt(Q_kp1'*Q_kp1);

% Gyro Bias Update
if SIM == 1
    % Constant Bias
    bg = X(5:7,k);
else
    % Random Walk Bias
    bg = X(5:7,k) + dt*X(11:13,k);
end

% Rebuild the State Vector for Each Iteration
Xx(:,k) = [Q_kp1_n; bg];

% Calculated the Weighted Mean of the State Vector
x_bar = x_bar + wm(k)*Xx(:,k);
end

% Calculate Error Covariance
Ax = sqrt(wc(2))*(Xx(:,2:n) - x_bar(:,ones(1,n-1)));
% QR Decomposision
[~, sx] = qr(Ax',0);
% Cholesky Update (Downdate for negative zeroth weight)
Sx = cholupdate(sx, sqrt(-wc(1))*(Xx(:,1)-x_bar), '-');

% Non-Linear Measurement Process Equations
Y = zeros(nR, n);                                % Pre-fill
y_bar = zeros(nR,1);                             % Pre-fill
Heading = zeros(1,n);                            % Pre-fill
if isempty(H1)
    H1 = zeros(1,n);                             % Pre-fill
    H2 = zeros(1,n);                             % Pre-fill
    Heading0 = zeros(1,n);                       % Pre-fill
    ii = zeros(1,n);                             % Pre-fill
end
for k = 1:n

if SIM == 1
    noise_R = X(11:13,k);
else
    noise_R = X(14:16,k);
end

% Quaternion to Euler Angles
q0 = Xx(1,k); q1 = Xx(2,k); q2 = Xx(3,k); q3 = Xx(4,k);
phi = atan2(2*(q2*q3+q0*q1), q0^2-q1^2-q2^2+q3^2) + noise_R(1);
theta = asin(-2*(q1*q3-q0*q2))+ noise_R(2);
psi = mod((atan2(2*(q1*q2+q0*q3), q0^2+q1^2-q2^2-q3^2))+ noise_R(3),
2*pi);
```

```
% Unwrap Heading Angle From [0, 2*pi] Range to Aviod Jumps
if nn == 1
    H1(k) = psi;
    Heading0(k) = H1(k);
    Heading(k) = H1(k);
else
    H2(k) = psi;
    if (H2(k)-H1(k)) <= -100*pi/180
        ii(k) = ii(k)+1;
    end
    if (H2(k)-H1(k)) >= 100*pi/180
        ii(k) = ii(k)-1;
    end
    Heading(k) = Heading0(k)+((H2(k)+2*pi*ii(k)) - Heading0(k));
    H1(k) = H2(k);
end

% Calculated Observations
Y(:,k)=[phi; theta; Heading(k)];

% Calculated Observation Mean
y_bar = y_bar + wm(k)*Y(:,k);
end

% Estimated Observation Covariance
Bx = sqrt(wc(2))*(Y(:,2:n) - y_bar(:,ones(1,n-1)));
% QR Decomposition
[~, sy] = qr(Bx',0); % **"qr" PRODUCES UPPER TRIANGULAR MATRIX**
% Cholesky Update (or Downdate)
Sy = cholupdate(sy, sqrt(-wc(1))*(Y(:,1)-y_bar), '-');
% ***Sy MUST BE LOWER TRIANGULAR***
Sy = Sy';

% Estimated Cross Covariance
Pxy = zeros(nx,nR);
for k = 1:n
    Pxy = Pxy + wc(k)*((Xx(:,k) - x_bar)*(Y(:,k) - y_bar)');
end

% Measurement Method 1 (Accelerometer-GPS-Compass Fusion)
% Rotate GPS XYZ Acceration about Heading
rx = ax_gps*cos(heading)+ay_gps*sin(heading);
ry = -ax_gps*sin(heading)+ay_gps*cos(heading);
rz = az_gps-g;
% Computer Pitch
theta = atan((-rx*rz - fx*sqrt(rx^2+rz^2-fx^2)) / (fx^2-rz^2));
% Compute Roll
r_theta = rx*sin(theta) + rz*cos(theta);
fc = fy-(norm(x_bar2(4:6)))*r; % Added Coriolis Term
phi = atan((r_theta*ry + fc*sqrt(ry^2+r_theta^2-fc^2)) / (fc^2-
r_theta^2));

% Heading Unwrap from [0, 2*pi]
if nn == 1
```

```
    H3 = heading;
    H4 = 0;
    psi0 = H3;
    psi_m = H3;
    jj=0;
else
    H4 = heading;
    if (H4-H3) <= -100*pi/180
        jj = jj+1;
    end
    if (H4-H3) >= 100*pi/180
        jj = jj-1;
    end
    psi_m = psi0+((H4+2*pi*jj) - psi0);
    H3 = H4;
end

% Measurement Vector
Z = [phi; theta; psi_m];

% Kalman Filter Equations
K = (Pxy/Sy')/Sy;                                   % Efficient Least
Squares
x_bar = x_bar + K*(Z - y_bar);                      % Kalman Correction

% Covariance Corrections
Ux = K*Sy;
for k = 1:nR
    Sx = cholupdate(Sx, Ux(:,k), '-');              % Cholesky Downdate
end
Sx = Sx';                                           % MUST BE LOWER
TRIANGULAR

nn = 2;
%% INS ESTIMATOR

% Build Augmented State Vector
X = zeros(nxa2, n2);
xa = [x_bar2; zeros(nxa2-nx2,1)];

% Build Augmented Covariance Matrix
Sxa2 = [Sx2 zeros(nx2,nQ2) zeros(nx2,nR2);
        zeros(nQ2,nx2) Qv2 zeros(nQ2,nR2);
        zeros(nR2,nx2) zeros(nR2,nQ2) R2];

% Construct n+2 SIGMA Vectors
for k = 1:n2
    X(:,k) = xa + Sxa2*sig2(:,k);
end

% Time Update
Xx = zeros(nx2, n2);                                % Pre-fill
x_bar2 = zeros(nx2, 1);                             % Pre-fill
for k = 1:n2
```

```
% Rotation Matrix
R_t2b = rot_t2b(x_bar(1:4));
R_b2t = R_t2b';

% Posittion Update
P_kp1 = X(1:3,k) + dt*[X(4:5,k); -X(6,k)];

% Velocity and Bias Update
if SIM == 1
% No Sidereal
    V_kp1 = X(4:6,k) + dt*(R_b2t*([fx; fy; fz] - X(7:9,k) - ...
            X(10:12, k)) + [0; 0; g]);
    ba_kp1 = X(7:9,k);
else
% Sidereal Included
    V_kp1 = X(4:6,k) + dt*(R_b2t*([fx; fy; fz] - X(7:9,k) - ...
            X(10:12, k)) + [0; 0; g] - 2*cross([wei*cos(lat0); ...
            0; -wei*sin(lat0)], X(4:6,k)));
    ba_kp1 = X(7:9,k) + dt*X(13:15,k);
end

% States
Xx(:,k) = [P_kp1; V_kp1; ba_kp1];

% Calculated Mean
x_bar2 = x_bar2 + wm2(k)*Xx(:,k);
end

% Predicted Error Covariance
Ax = sqrt(wc2(2))*(Xx(:,2:n2) - x_bar2(:,ones(1,n2-1)));
% QR Decomposition
[~, Sx2] = qr(Ax',0);
% Cholesky Factor Update
Sx2 = cholupdate(Sx2, sqrt(-wc2(1))*(Xx(:,1)-x_bar2), '-');

% Measurement Equations
Y = zeros(nR2, n2);                                  % Pre-fill
y_bar = zeros(nR2,1);                                % Pre-fill
for k = 1:n2

if SIM == 1
    % Calculated Observations
    Y(:,k)=Xx(1:6,k)+X(13:18,k);
else
    % Calculated Observations
    Y(:,k)=Xx(1:6,k)+X(16:21,k);
end

% Calculated Observation Mean
y_bar = y_bar + wm2(k)*Y(:,k);
end

% Estimated Observation Covariance
Bx = sqrt(wc2(2))*(Y(:,2:n2) - y_bar(:,ones(1,n2-1)));
```

```
% QR Decomposition
[~, Sy] = qr(Bx',0);
% Cholesky Factor Update
Sy = cholupdate(Sy, sqrt(-wc2(1))*(Y(:,1)-y_bar), '-');
% Make Sy Lower Triangular
Sy = Sy';

% Estimated Cross Covariance
Pxy = zeros(nx2,nR2);
for k = 1:n2
    Pxy = Pxy + wc2(k)*((Xx(:,k) - x_bar2)*(Y(:,k) - y_bar)');
end

% Measurements
Z = [N_sat; E_sat; D_sat; speed*cos(CoG); speed*sin(CoG); Vd_sat];

% Kalman Filter Equations
% Kalman Gain w/ Efficient Least Squares
K = (Pxy/Sy')/Sy;
% Kalman State Corrections
x_bar2 = x_bar2 + K*(Z - y_bar);
% Covariance Correction
Ux = K*Sy;
for k = 1:nR2
    Sx2 = cholupdate(Sx2, Ux(:,k), '-');
end
% Make Sx2 Lower Right Trangular
Sx2 = Sx2';

X_HAT = [x_bar(1:4); x_bar2(1:6); x_bar(5:7); x_bar2(7:9)];
end
kk=kk+1;
end

function R_t2b = rot_t2b(x)
q0 = x(1); q1 = x(2); q2 = x(3); q3 = x(4);
R_t2b = [q0^2+q1^2-q2^2-q3^2 2*(q1*q2+q0*q3) 2*(q1*q3-q0*q2);
         2*(q2*q1-q0*q3) q0^2-q1^2+q2^2-q3^2 2*(q2*q3+q0*q1);
         2*(q3*q1+q0*q2) 2*(q3*q2-q0*q1) q0^2-q1^2-q2^2+q3^2];
end

function q = e2q(phi, theta, psi)
% Form Rotation Matrix
R_psi = [cos(psi) sin(psi) 0; -sin(psi) cos(psi) 0; 0 0 1];
R_theta = [cos(theta) 0 -sin(theta); 0 1 0; sin(theta) 0 cos(theta)];
R_phi = [1 0 0; 0 cos(phi) sin(phi); 0 -sin(phi) cos(phi)];
R_n2b = R_phi*R_theta*R_psi;

% Extract Quaternions
q0=sqrt(1+trace(R_n2b))/2;
q1=(R_n2b(2,3)-R_n2b(3,2))/(4*q0);
q2=(R_n2b(3,1)-R_n2b(1,3))/(4*q0);
q3=(R_n2b(1,2)-R_n2b(2,1))/(4*q0);
```

```
q = [q0; q1; q2; q3];
end
```

2. Measurement Model Two Implementation

```
function X_HAT = SRSSUKF2(u)
%% SQUARE ROOOT SPHERICAL SIMPLEX UNSCENTED KALMAN FILTER
% Measurement Model 2
% LT Steven Terjesen
% September 2014

% This estimator is built for use in MATLAB Simulink. There are 39
inputs
% required and can be run in Simulink with an “Interpolated MATLAB
% Function” block. The first 13 inputs are the vehicle data: Course
Over
% Ground, Speed Over Ground, Accelerometer Measurements, Gyro
Measurements,
% GPS Measurements (In LTP XNorth-YEast-ZDown), Heading Measurement,
and
% Vertical Velocity (zeroed out for SEAFOXII data). Inputs 14 through
35
% are the Process Noise and Measurement Noise diagonal elements. These
% elementes are not hard coded to allow for easier tuning. Inputs 36
% through 38 are the N-E-D accelerations in the LTP frame. Input 39 is
a
% toggle for selecting whether the data is from Condor or SEAFOX II.

%% System Inputs
% Measurement inputs
CoG = u(1);                                          %GPS Course Over Ground
speed = u(2);                                        %GPS Speed Over Ground
fx = u(3); fy = u(4); fz = u(5);                     %IMU Accelerations
p = u(6); q = u(7); r = u(8);                        %IMU Angular Rates
N_sat = u(9); E_sat = u(10); D_sat = u(11);          %GPS Position (NED)
heading = u(12);                                     %GPS Heading
Vd_sat = u(13);                                      %GPS LTP Down Velocity

% Process and Measurement Noise Matrices
R = chol(diag(u(14:17)));                            %AHRS Measurement Noise
R2 = chol(diag(u(18:23)));                           %INS Measurement Noise

SIM = u(39);                                         %Condor(1) or
SEAFOXII(2)
%                                                     Simulation Selector

if SIM == 1
    Qv = chol(diag(u(24:26)));                       %AHRS Process Noise
Condor
    Qv2 = chol(diag(u(30:32)));                      %INS Process Noise
Condor
else
    Qv = chol(diag(u(24:29)));                       %AHRS Process Noise
SEAFOX
```

```
    Qv2 = chol(diag(u(30:35)));                 %INS Process Noise
SEAFOX
    Vd_sat = 0;                                 %No Vertical Velocity
%                                                measurement available
on
%                                                SEAFOX II, set Vd=0.
    D_sat = 0;                                  %No Altitude
Measurement
%                                                available on SEAFOXII,
set
%                                                D_sat = 0;
end

% GPS Accelerations
ax_gps = u(36); ay_gps = u(37); az_gps = u(38); %GPS Accelerations as
                                                % calculated from 3rd
                                                % Order Filter


%% Initialization
persistent x_bar Heading0 H1 H2 ii jj psi0 H3 H4 ...
Sx Sx2 kk dt g nn nx nxa nQ nR n wc wm x_bar2 nx2 nxa2 n2 nQ2 nR2 wc2
...
wm2 sig sig2 wei lat0

% Allows time for Condor To steady out during live testing
if isempty(kk)
    kk = 0;
end
if kk < 1 && SIM==1
    X_HAT = zeros(16,1);
else

if isempty(x_bar)
% Miscellaneous
dt = 0.01;                                      %Filter dt [sec]
g = 9.8;                                        %Gravity Constant
[m/s^2]
wei = 7.292115*10^-5;                           %Sidereal Rate [rad/s]
lat0= 0.639268394832413;                        %Origin of LTP [rad]
%lon0 = -2.115435878466264
nn=1;                                           %Heading Count
Initializer

% AHRS Initialization
% Euler Angle Initialization
rx = ax_gps*cos(heading)+ay_gps*sin(heading);
ry = -ax_gps*sin(heading)+ay_gps*cos(heading);
rz = az_gps-g;
theta_0 = atan((-rx*rz - fx*sqrt(rx^2+rz^2-fx^2))/(fx^2-rz^2));
r_theta = rx*sin(theta_0) + rz*cos(theta_0);
fc = fy-speed*r;
phi_0 = atan((r_theta*ry + fc*sqrt(ry^2+r_theta^2-fc^2))/(fc^2-
r_theta^2));
psi_0=heading;
```

```
% Euler Angles to Quaternions
q_0 = e2q(phi_0, theta_0, psi_0);
% Gyro Bias Initial Values
% bg_0 = [2.4e-4; -1e-4; 2e-4];
bg_0 = [0; 0; 0];
% Inital State Vector
x_bar = [q_0; bg_0];
% Initial Covariance Estimate
Px = diag([1e-5*ones(1,4) 1e-3*ones(1,3)]);
% Px = diag([1e-5*ones(1,4) 1e-6*ones(1,3)]);
% Inital Matrix Square Root
Sx = chol(Px);

%INS Initialization
% Position & Velocity Initialization
x_0 = N_sat;
y_0 = E_sat;
z_0 = D_sat;
vn_0 = speed*cos(CoG);
ve_0 = speed*sin(CoG);
vd_0 = Vd_sat;
% Accelerometer Initial Bias Estimate
ba_0 = [0; 0; 0];
% Initial State Vector
x_bar2 = [x_0; y_0; z_0; vn_0; ve_0; vd_0; ba_0];
% Initial Covariance Estimate
Px2 = diag([1e-5*ones(1,3), 1e-5*ones(1,3), 1e-3*ones(1,3)]);
% Inital Matrix Square Root
Sx2 = chol(Px2);

% AHRS SIGMA Weights Initialization
nx = length(x_bar);                              % Number of AHRS States
nQ = length(diag(Qv));                           % Process Noise States
nR = length(diag(R));                            % Measurement Noise
States
nxa = nx+nQ+nR;                                  % Total Augmented
States
n = nxa+2;                                       % Number of Iterations
alpha = 1e-3;                                    % Tunable Scaleing
Factor
beta = 2;                                        % beta = 2 for Gaussian
PDF
Wx0 = 1/3;                                       % Tunable, 0<W<1
Wx = zeros(n,1);                                 % Pre-fill
% Simplex Weights
for k = 1:n
    if k == 1
        Wx(k) = Wx0;
    else
        Wx(k) = (1-Wx0)/(nxa+1);
    end
end
% Scaled Simplex Weights
for k = 1:n
    if k == 1
```

```
        wx(k)= 1+(Wx(k)-1)/alpha^2;
    else
        wx(k) = Wx(k)/alpha^2;
    end
end
% Incorperate prior PDF knowledge for Higher Order Moments
for k = 1:n
    if k == 1
        wm(k) = wx(k);
        wc(k) = wx(k)+(1-alpha^2+beta);
    else
        wm(k) = wx(k);
        wc(k) = wx(k);
    end
end

%Sigma Matrix
sig = zeros(nxa, n);                                % Pre-fill
sig(1,1) = 0;                                       % Initialize (1,1)
sig(1,2) = -1/sqrt(2*wx(2));                        % Initialize (1,2)
sig(1,3) = 1/sqrt(2*wx(2));                         % Initialize (1,3)
for j = 2:nxa
    for i = 1:1
        sig(j,i) = 0;
    end
    for i = 2:j+1
        sig(j,i) = -1/sqrt(j*(j+1)*wx(2));
    end
    for i = j+2:j+2
        sig(j,i) = j/sqrt(j*(j+1)*wx(2));
    end
end

% INS SIGMA Weights Initialization
nx2 = length(x_bar2);                               % Number of INS States
nQ2 = length(diag(Qv2));                            % Process Noise States
nR2 = length(diag(R2));                             % Measurement Noise
States
nxa2 = nx2+nQ2+nR2;                                 % Total Augmented
States
n2 = nxa2+2;                                        % Total Iterations
alpha = 1e-3;                                       % Tunable, 0<alpha<1
beta = 2;                                           % beta = 2 for Gaussian
PDF
Wx02 = 1/3;                                         % Tunable, 0<W<1
Wx2 = zeros(n2,1);                                  % Pre-fill

% Sigma Weights
for k = 1:n2
    if k == 1
        Wx2(k) = Wx02;
    else
        Wx2(k) = (1-Wx02)/(nxa2+1);
    end
end
```

```
% Scaled Sigma Weights
for k = 1:n2
    if k == 1
        wx2(k)= 1+(Wx2(k)-1)/alpha^2;
    else
        wx2(k) = Wx2(k)/alpha^2;
    end
end

% Incorperate prior PDF knowledge for Higher Order Moments
for k = 1:n2
    if k == 1
        wm2(k) = wx2(k);
        wc2(k) = wx2(k)+(1-alpha^2+beta);
    else
        wm2(k) = wx2(k);
        wc2(k) = wx2(k);
    end
end

% Sigma Matrix
sig2 = zeros(nxa2, n2);                          % Pre-fill
sig2(1,1) = 0;                                   % Initialize (1,1)
sig2(1,2) = -1/sqrt(2*wx2(2));                   % Initialize (1,2)
sig2(1,3) = 1/sqrt(2*wx2(2));                    % Initialize (1,3)
for j = 2:nxa2
    for i = 1:1
        sig2(j,i) = 0;
    end
    for i = 2:j+1
        sig2(j,i) = -1/sqrt(j*(j+1)*wx2(2));
    end
    for i = j+2:j+2
        sig2(j,i) = j/sqrt(j*(j+1)*wx2(2));
    end
end

end
%% AHRS ESTIMATOR

% Build Augmented State Vector
X = zeros(nxa, n);
xa = [x_bar; zeros(nxa-nx,1)];

% Build Augmented Covariance Matrix
Sxa = [Sx zeros(nx,nQ) zeros(nx,nR);
        zeros(nQ,nx) Qv zeros(nQ,nR);
        zeros(nR,nx) zeros(nR,nQ) R];

% Build n+2 Sigma Vectors
for k = 1:n
    X(:,k) = xa + Sxa*sig(:,k);
end
```

```
% Time Update
Xx = zeros(nx, n);
x_bar = zeros(nx, 1);
for k = 1:n

% Gyro Model  w_bi(true) = w_bi(measured) - bias - noise
w_bi_b = [p-X(5,k)-X(8,k); q-X(6,k)-X(9,k); r-X(7, k)-X(10,k)];

if SIM == 1
    % No Sidereal Rate
    w_bt = w_bi_b;
else
    % Sidereal Rate Included
    R_t2b = rot_t2b(X(1:4,k));
    w_bt = w_bi_b - R_t2b*[wei*cos(lat0); 0; -wei*sin(lat0)];
end

% Quaternion Update
w1 = w_bt(1); w2 = w_bt(2); w3 = w_bt(3);
Q_kp1 = X(1:4, k) + (dt/2)*[0,  -w1,  -w2,  -w3;
                            w1,  0,    w3,  -w2;
                            w2, -w3,   0,    w1;
                            w3,  w2,  -w1,   0]*X(1:4, k);

% Quaternion Normalization
Q_kp1_n = Q_kp1/sqrt(Q_kp1'*Q_kp1);

% Gyro Bias Update
if SIM == 1
    % Constant Bias
    bg = X(5:7,k);
else
    % Random Walk Bias
    bg = X(5:7,k) + dt*X(11:13,k);
end

% Rebuild the State Vector for Each Iteration
Xx(:,k) = [Q_kp1_n; bg];

% Calculated the Weighted Mean of the State Vector
x_bar = x_bar + wm(k)*Xx(:,k);
end

% Calculate Error Covariance
Ax = sqrt(wc(2))*(Xx(:,2:n) - x_bar(:,ones(1,n-1)));
% QR Decomposision
[~, sx] = qr(Ax',0);
% Cholesky Update (Downdate for negative zeroth weight)
Sx = cholupdate(sx, sqrt(-wc(1))*(Xx(:,1)-x_bar), '-');

% Non-Linear Measurement Process Equations
Y = zeros(nR, n);                                % Pre-fill
```

```
y_bar = zeros(nR,1);                                % Pre-fill
Heading = zeros(1,n);                               % Pre-fill
if isempty(H1)
    H1 = zeros(1,n);                                % Pre-fill
    H2 = zeros(1,n);                                % Pre-fill
    Heading0 = zeros(1,n);                          % Pre-fill
    ii = zeros(1,n);                                % Pre-fill
end
for k = 1:n

% Rotation Matrix (LTP to Body)
R_t2b = rot_t2b(Xx(1:4,k));
% Gyro Measurements
w_bi = [p; q; r] - Xx(5:7,k);
if SIM == 1
    noise_R = X(11:14,k);
    w_bt = w_bi;
else
    noise_R = X(14:17,k);
    w_bt = w_bi - R_t2b*[wei*cos(lat0); 0; -wei*sin(lat0)];
end

% Quaternion to Euler Angles
q0 = Xx(1,k); q1 = Xx(2,k); q2 = Xx(3,k); q3 = Xx(4,k);
psi = mod(atan2(2*(q1*q2+q0*q3), q0^2+q1^2-q2^2-q3^2), 2*pi)+
noise_R(4);
% Unwrap Heading Angle From [0, 2*pi] Range to Aviod Jumps
if nn == 1
    H1(k) = psi;
    Heading0(k) = H1(k);
    Heading(k) = H1(k);
else
    H2(k) = psi;
    if (H2(k)-H1(k)) <= -100*pi/180
        ii(k) = ii(k)+1;
    end
    if (H2(k)-H1(k)) >= 100*pi/180
        ii(k) = ii(k)-1;
    end
    Heading(k) = Heading0(k)+((H2(k)+2*pi*ii(k)) - Heading0(k));
    H1(k) = H2(k);
end

% Accelerometer Estimate
FB_hat = R_t2b*[ax_gps; ay_gps; az_gps] + ...
        cross(w_bt, R_t2b*x_bar2(4:6)) - ...
        R_t2b*[0; 0; g]-x_bar2(7:9)-noise_R(1:3);


% Calculated Observations
Y(:,k)=[FB_hat; Heading(k)];

% Calculated Observation Mean
y_bar = y_bar + wm(k)*Y(:,k);
```

```
end

% Estimated Observation Covariance
Bx = sqrt(wc(2))*(Y(:,2:n) - y_bar(:,ones(1,n-1)));
% QR Decomposition
[~, sy] = qr(Bx',0); % **"qr" PRODUCES UPPER TRIANGULAR MATRIX**
% Cholesky Update (or Downdate)
Sy = cholupdate(sy, sqrt(-wc(1))*(Y(:,1)-y_bar), '-');
% ***Sy MUST BE LOWER TRIANGULAR***
Sy = Sy';

% Estimated Cross Covariance
Pxy = zeros(nx,nR);
for k = 1:n
    Pxy = Pxy + wc(k)*((Xx(:,k) - x_bar)*(Y(:,k) - y_bar)');
end

% Measurement Method 2 (Accelerometer)

% Heading Unwrap from [0, 2*pi]
if nn == 1
    H3 = heading;
    H4 = 0;
    psi0 = H3;
    psi_m = H3;
    jj=0;
else
    H4 = heading;
    if (H4-H3) <= -100*pi/180
        jj = jj+1;
    end
    if (H4-H3) >= 100*pi/180
        jj = jj-1;
    end
    psi_m = psi0+((H4+2*pi*jj) - psi0);
    H3 = H4;
end

% Measurement Vector
Z = [fx; fy; fz; psi_m];

% Kalman Filter Equations
K = (Pxy/Sy')/Sy;                                % Efficient Least
Squares
x_bar = x_bar + K*(Z - y_bar);                   % Kalman Correction

% Covariance Corrections
Ux = K*Sy;
for k = 1:nR
    Sx = cholupdate(Sx, Ux(:,k), '-');           % Cholesky Downdate
end
Sx = Sx';                                        % MUST BE LOWER
TRIANGULAR
```

```
nn = 2;
%% INS ESTIMATOR

% Build Augmented State Vector
X = zeros(nxa2, n2);
xa = [x_bar2; zeros(nxa2-nx2,1)];

% Build Augmented Covariance Matrix
Sxa2 = [Sx2 zeros(nx2,nQ2) zeros(nx2,nR2);
        zeros(nQ2,nx2) Qv2 zeros(nQ2,nR2);
        zeros(nR2,nx2) zeros(nR2,nQ2) R2];

% Construct n+2 SIGMA Vectors
for k = 1:n2
    X(:,k) = xa + Sxa2*sig2(:,k);
end

% Time Update
Xx = zeros(nx2, n2);                                    % Pre-fill
x_bar2 = zeros(nx2, 1);                                 % Pre-fill
for k = 1:n2
% Rotation Matrix
R_t2b = rot_t2b(x_bar(1:4));
R_b2t = R_t2b';

% Posittion Update
P_kp1 = X(1:3,k) + dt*[X(4:5,k); -X(6,k)];

% Velocity and Bias Update
if SIM == 1
% No Sidereal
    V_kp1 = X(4:6,k) + dt*(R_b2t*([fx; fy; fz] - X(7:9,k) - ...
            X(10:12, k)) + [0; 0; g]);
    ba_kp1 = X(7:9,k);
else
% Sidereal Included
    V_kp1 = X(4:6,k) + dt*(R_b2t*([fx; fy; fz] - X(7:9,k) - ...
            X(10:12, k)) + [0; 0; g] - 2*cross([wei*cos(lat0); ...
            0; -wei*sin(lat0)], X(4:6,k)));
    ba_kp1 = X(7:9,k) + dt*X(13:15,k);
end

% States
Xx(:,k) = [P_kp1; V_kp1; ba_kp1];

% Calculated Mean
x_bar2 = x_bar2 + wm2(k)*Xx(:,k);
end

% Predicted Error Covariance
Ax = sqrt(wc2(2))*(Xx(:,2:n2) - x_bar2(:,ones(1,n2-1)));
% QR Decomposition
[~, Sx2] = qr(Ax',0);
% Cholesky Factor Update
```

```
Sx2 = cholupdate(Sx2, sqrt(-wc2(1))*(Xx(:,1)-x_bar2), '-');

% Measurement Equations
Y = zeros(nR2, n2);                                    % Pre-fill
y_bar = zeros(nR2,1);                                  % Pre-fill
for k = 1:n2

if SIM == 1
    % Calculated Observations
    Y(:,k)=Xx(1:6,k)+X(13:18,k);
else
    % Calculated Observations
    Y(:,k)=Xx(1:6,k)+X(16:21,k);
end

% Calculated Observation Mean
y_bar = y_bar + wm2(k)*Y(:,k);
end

% Estimated Observation Covariance
Bx = sqrt(wc2(2))*(Y(:,2:n2) - y_bar(:,ones(1,n2-1)));
% QR Decomposition
[~, Sy] = qr(Bx',0);
% Cholesky Factor Update
Sy = cholupdate(Sy, sqrt(-wc2(1))*(Y(:,1)-y_bar), '-');
% Make Sy Lower Triangular
Sy = Sy';

% Estimated Cross Covariance
Pxy = zeros(nx2,nR2);
for k = 1:n2
    Pxy = Pxy + wc2(k)*((Xx(:,k) - x_bar2)*(Y(:,k) - y_bar)');
end

% Measurements
Z = [N_sat; E_sat; D_sat; speed*cos(CoG); speed*sin(CoG); Vd_sat];

% Kalman Filter Equations
% Kalman Gain w/ Efficient Least Squares
K = (Pxy/Sy')/Sy;
% Kalman State Corrections
x_bar2 = x_bar2 + K*(Z - y_bar);
% Covariance Correction
Ux = K*Sy;
for k = 1:nR2
    Sx2 = cholupdate(Sx2, Ux(:,k), '-');
end
% Make Sx2 Lower Right Trangular
Sx2 = Sx2';

X_HAT = [x_bar(1:4); x_bar2(1:6); x_bar(5:7); x_bar2(7:9)];
end
kk=kk+1;
end
```

```
function R_t2b = rot_t2b(x)
q0 = x(1); q1 = x(2); q2 = x(3); q3 = x(4);
R_t2b = [q0^2+q1^2-q2^2-q3^2 2*(q1*q2+q0*q3) 2*(q1*q3-q0*q2);
         2*(q2*q1-q0*q3) q0^2-q1^2+q2^2-q3^2 2*(q2*q3+q0*q1);
         2*(q3*q1+q0*q2) 2*(q3*q2-q0*q1) q0^2-q1^2-q2^2+q3^2];
end

function q = e2q(phi, theta, psi)
% Form Rotation Matrix
R_psi = [cos(psi) sin(psi) 0; -sin(psi) cos(psi) 0; 0 0 1];
R_theta = [cos(theta) 0 -sin(theta); 0 1 0; sin(theta) 0 cos(theta)];
R_phi = [1 0 0; 0 cos(phi) sin(phi); 0 -sin(phi) cos(phi)];
R_n2b = R_phi*R_theta*R_psi;

% Extract Quaternions
q0=sqrt(1+trace(R_n2b))/2;
q1=(R_n2b(2,3)-R_n2b(3,2))/(4*q0);
q2=(R_n2b(3,1)-R_n2b(1,3))/(4*q0);
q3=(R_n2b(1,2)-R_n2b(2,1))/(4*q0);

q = [q0; q1; q2; q3];
end
```

LIST OF REFERENCES

[1] "The Navy unmanned surface vehicle (USV) master plan," U.S. Dept. of the Navy, Washington, DC, 23 Jul 2007. [Online]. Available: http://www.navy.mil/navydata/technology/usvmppr.pdf. Accessed Aug. 05, 2014.

[2] M. A. Hurban, "Adaptive speed controller for the SeaFox autonomous surface vessel," M.S. thesis, Dept. Mech. and Aerospace Eng., Naval Postgraduate School, Monterey, CA, 2012.

[3] W. McAuley, *Seaborne Controller Area Network (SEACAN) System for the High Speed Maneuverable Surface Target (HSMST) Operations and Maintenance (O&M) Manual*, Naval Air Warface Centero Weapons Division, Port Hueneme, CA, 2007.

[4] Vector G2 Satellite Compass. (n.d.). ComNav Marine Ltd. [Online]. Available: http://www.comnavmarine.com/html/cmnv2341.htm. Accessed Sep. 19, 2014.

[5] *User's Manual for the HG1700AH58 Inertial Measurement Unit*, Minneapolis, MN: Honeywell International Inc., 2006.

[6] Robot Operating System (ROS). (n.d.). Open Source Robitcs Foundation. [Online]. Available: http://www.ros.org/about-ros/. Accessed Sep. 19, 2014.

[7] R. E. Kalman, "A new approach to linear filtering and prediction problems," *ASME-Journal of Basic Engineering*, vol. 82 (Series D), pp. 35-45, 1960.

[8] J. A. Farrell, *Aided Navigation: GPS with High Rate Sensors*, New York: McGraw-Hill, 2008.

[9] "Kalman Filtering from Singal Processing Standpoint," unpublished class notes for Guidance Navigation and Control of Marine Systems, Dept. of Mechanical and Aerospace Engineering, Naval Postgraduate School, Monterey, CA, Fall 2014.

[10] N. J. Gordon, D. J. Salmond and A. F. Smith, "Novel approach to nonlinear/non-Gaussian Bayesian state estimation," *IEEE Proceedings*, vol. 140, no. 2, pp. 107–13, 1993.

[11] S. Julier, J. Uhlmann and H. Durrant-Whyte, "A new method for the nonlinear trasnformation of means and covariance in filters and estimators," *IEEE Transactions on Automatic Control*, vol. 45, no. 3, pp. 477–82, 2000.

[12] S. J. Julier and J. K. Uhlmann, “Unscented filtering and nonlinear estimation,” *Proceedings of the IEEE*, vol. 92, no. 3, pp. 401–21, 2004.

[13] E. Wan and R. Merwe, “The unscented Kalman filter for nonlinear estimation,” in *Adaptive Systems for Signal Proceessing, Communicaitons, and Control Symposium 2000*, Lake Louise, Alta, 2000.

[14] R. Merwe and E. A. Wan, “The square-root unscented Kalman filter for state and parameter-estimation,” in *2001 IEEE International Conference on Acoustics, Speech, and Signal Processing*, Salt Lake City, UT, 2001.

[15] S. Julier, “The spherical simplex unscented transformation,” in *Proceedings of the 2003 American control Conference*, Denver, CO, 2003.

[16] X. Tang, X. Zhao and X. Zhang, “The square-root spherical simplex unscented Kalman filter for state and parameter estimation,” in *9th International Conference on Signal Processing*, Beijing, China, 2008.

[17] Condor: The Competition Soaring Simulator. (n.d.). Condor Team. [Online]. Available: http://www.condorsoaring.com/. Accessed Sep. 19, 2014.

[18] T. Fossen, *Mathematical Models for Control of Aircraft and Satellites*, Trondheim, Norway: Dept. of Engineering Cybernetics, NTNU, 2011.

[19] G. M. Siouris, *Aerospace Avionics Systems: A Modern Sysnthesis*, San Diego, CA: Academic Press, 1993.

[20] H. Lopes, E. Kampen, and Q. Chu, “Attitude determindation of highly dynamic fixed-wing UAVs with GPS/MEMS-AHRS integration,” *AIAA Guidance, Navigation, and Control Conferences*, Minneapolis, MN, 2011.

[21] D. Kingston, “Implementation issues of real-time trajectory generation on small UAVs,” M.S. thesis, Dept. of Electrical and Computer Eng., Bringham Young University, Provo, UT, 2004.

[22] A. Eldredge, “Improved state estimation for miniature air vehicles,” M.S. thesis, Dept. of Mech. Eng., Bringham Young University, Provo, UT, 2006.

[23] J. Diebel, *Representing Attitude: Euler Angles, Unit Quaternions, and Rotation Vectors*, Palo Alto, CA: Stanford University, 2006.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California